Computationally Bounded Robust Compilation and Universally Composable Security

Robert Künneman
CISPA Helmholtz Center for Information Security
robert.kuennemann@cispa.de

Marco Patrignani
University of Trento
marco.patrignani@unitn.it

Ethan Cecchetti
University of Wisconsin–Madison*
cecchetti@wisc.edu

***Abstract*—Universal Composability (UC) is the gold standard for cryptographic security, but mechanizing proofs of UC is notoriously difficult. A recently-discovered connection between UC and Robust Compilation (RC)—a novel theory of secure compilation—provides a means to verify UC proofs using tools that mechanize equality results. Unfortunately, the existing methods apply only to perfect UC security, and real-world protocols relying on cryptography are only *computationally* secure.**

This paper addresses this gap by lifting the connection between UC and RC to the computational setting, extending techniques from the RC setting to apply to computational UC security. Moreover, it further generalizes the UC–RC connection beyond computational security to arbitrary equalities, providing a framework to subsume the existing perfect case, and to instantiate future theories with more complex notions of security. This connection allows the use of tools for proofs of computational indistinguishability to properly mechanize proofs of computational UC security. We demonstrate this power by using CryptoVerif to mechanize a proof that parts of the Wireguard protocol are computationally UC secure. Finally, all proofs of the framework itself are verified in Isabelle/HOL.

I. Introduction

In cryptography, universal composability (UC) [16] is a framework for the specification and analysis of cryptographic protocols with a key guarantee about compositionality [15, 36, 53]: If a protocol is UC-secure, it behaves like some high-level, secure-by-construction ideal functionality no matter what the protocol interacts with. If that protocol is then used as a building block inside a larger protocol, it is safe to replace the smaller protocol with its ideal functionality when reasoning about the security of the larger protocol. In other words, UC protocols are secure even when composed with larger protocols.

Proving that a protocol attains UC security is notoriously complex and error-prone, so it is important to provide ways to mechanize these proofs. Patrignani et al. [43] recently identified a simple and scalable way to define proofs of UC by relying on a surprising connection between UC and Robust Compilation (RC).

RC is a hierarchy of criteria for secure compilation introduced by Abate et al. [2, 3]. The criteria describe the security of a compiler by which (hyper)properties [21] it preserves.

Patrignani et al. [43] identify that UC security is deeply connected to Robust Hyperproperty-Preserving Compilation (RHC), the RC requirement that a compiler preserve arbitrary hyperproperties. Unfortunately, the connection they identify considers only *perfect* UC security, where the protocol and ideal functionality exhibit *identical* behaviors. Real cryptographic protocols are almost never perfectly secure, they rely on cryptographic primitives whose security depends on computational hardness assumptions. As a result, real "UC-secure" protocols rely on a *computational* definition of UC security, which allows the protocol to behave differently from the ideal functionality, but only in ways that are indistinguishable to a computationally bounded adversary. Since the results of Patrignani et al. [43] apply only to the perfect case, they provide little help in verifying the far more prevalent proofs of computational UC security.

For example, consider the following single-bit commitment protocol due to Canetti and Fischlin [17]. To commit to bit b, generate a $4n$-bit pseudo-random value pr, and output pr if $b = 0$ and $pr \oplus \sigma$ if $b = 1$ where σ is a public $4n$-bit truly random value. This protocol does not perfectly UC emulate a functionality that simply indicates a bit has been committed by does nothing else before opening. An *unbounded* attacker can simply check if the commitment is one of the (exponentially many) possible pseudo-random values and correctly guess the value of b with overwhelming probability. The protocol is, however, *computationally* UC secure given a pseudo-random generator with the right structure [17], a proof that relies on a *polynomial* adversary's inability to distinguish pr from a truly random value.

This work addresses the limitation of Patrignani et al. [43] by lifting their results to the computational case, allowing us to consider the security of protocols like Canetti and Fischlin's commitment. To accomplish this goal, we replace equality of behaviors with *computational indistinguishability* in both the UC and RC theories. Making this switch in both the UC and RC contexts produces a notion of robust compilation that corresponds precisely to the already-established definition of computational UC security.

On the RC side, the change requires two fundamental modifications. First, Patrignani et al. [43] describe programs as producing probability distributions over possible traces and consider a protocol (or compiler) secure if the ideal functionality (source program) and protocol (compiled program) produce identical distributions. Computational indistinguishability, however, does not relate individual distributions. It relates *families* of distributions $\{X_n\}$ and $\{Y_n\}$ indexed by

*Work done in part while author was at the University of Maryland.

a security parameter n, and requires an adversary's ability to distinguish between X_n and Y_n to shrink quickly as n grows. We thus expand the definition of program behavior to include explicit security parameters. Second, RC definitions in general define robustness for all programs against all adversaries, while computational UC-security concerns only *polynomial-time* programs and attackers. We therefore extend the RC framework to consider specific classes of protocols and attackers.

With these two modifications to the RC theory of Abate et al. [2], we can define a new class of hyperproperties, CH, where each hyperproperty is the set of behaviors that are computationally indistinguishable from some ideal functionality. We also define a notion of Computationally-Robust Hyperproperty-Preserving Compilation ($CRHC$), a new notion of RC that preserves CH against polynomial-time attackers. Finally, we prove that computational UC security is equivalent to $CRHC$.

Technically, proving $CRHC$ amounts to proving that some source program (an ideal functionality), linked with an existentially bound attacker (a simulator) is computationally indistinguishable from a target program (a protocol). Fortunately, existing tools such as CryptoVerif [13] provide ways to mechanize such proofs. Thus, we showcase the ability to provide scalable proofs of computational UC via proofs of $CRHC$ by using CryptoVerif to mechanize a proof of computational indistinguishability for the Wireguard protocol [37].

Notably, the results of lifting the connection between UC and RC to the computational case contain nothing specific about computational indistinguishability and polynomial time. As a result, we are able to substantially generalize the theory by using any equivalence $\equiv$ defining indistinguishable behaviors, and an arbitrary predicate Q on programs and contexts to define the class of programs and contexts. Doing so produces a notion of UC security up-to $\equiv$, an RC notion of Q-robust preservation of $\equiv$-hyperproperties, and a proof that the two are equivalent. This result immediately subsumes the original connection of Patrignani et al. [43], using $=$ as $\equiv$ and the trivial predicates allowing all programs and contexts, our lifted result, using computational indistinguishability as the equivalence and polynomial time as the predicate, and suggests more definitions and connections to explore.

To summarize, the main contributions of this paper are:

- Section III explicitly models security parameters and computational indistinguishability in the RC framework, and uses these structures to extend the result of Patrignani et al. [43] to computational security.
- Section IV generalizes the computational result to arbitrary indistinguishability relations, making perfect security, computational security, and many other interesting equivalences special cases of a general theorem.
- Section VI uses these results to mechanize the proof of UC for the WireGuard protocol using the CryptoVerif tool, which provides computational security guarantees.

The rest of this paper provides background notions related to the main results (Section II), interesting details of the proofs of our main theorems (Section V), the presentation of related work (Section VII), and conclusions (Section VIII).

All theorems in Sections III, IV, and V are verified in the Isabelle/HOL theorem prover [32], while theorems in Section VI are verified in CryptoVerif [13]. The proof development is available at: https://uc-is-sc.github.io/.

II. Background

This section presents relevant background notions on Universal Composability (Section II-A), Robust Compilation (Section II-B) and their connection (Section II-C).

For aide reading [40], throughout the paper we will use *blue italic* for functionalities and source programs, **bold red** for protocols and target programs, and black for terms not specific to either context.

A. Universally Composable Security

Universally Composable (UC) security [16] defines a security notion that combines functional correctness and privacy and is both transitive as well as closed under protocol composition. This is achieved by refinement: a "secure" protocol is one that is "at least as secure" as a protocol where all parties forward their communication to a single entity that (a) computes the correct output (which they forward to the environment) and (b) leaks only the minimal amount of information via the network (which is conservatively modelled by sending this information to the attacker). There is a distinction between the environment, which models the trusted input and output from either higher-level protocols or the user of the system, and the attacker, which models the hostile network. Together, the environment $\mathcal{Z}$, protocol π and attacker $\mathbf{A}$ constitute a system that can be executed. The execution ends when the environment decides whether it is interacting with the real protocol or an ideal simulation and outputs a final bit to indicate its guess. Let $\textsc{Exec}(\mathcal{Z}, \mathbf{A}, \pi)$ be a random variable describing the outcome of this probabilistic process.

The strongest notion of "at least as secure as" says that any attack on a protocol can be simulated using the functionality. That is, only knowing the "acceptable" leakage built into the functionality is enough to convincingly reproduce any real attacker's behavior. For an encryption functionality, for instance, this leakage is the message length.

If this simulation can always produce *identical* behavior to the real attacker, the protocol is said to *perfectly* emulate the ideal functionality.

Definition 1 (Perfect UC Emulation)**.** A protocol π *perfectly* UC*-emulates* a functionality F, denoted $\pi \vdash_{\text{UC}}^{\equiv} F$ if, for all (unbounded) adversaries $\mathbf{A}$, there is a simulator S, such that for all environments $\mathcal{Z}$,

$$\textsc{Exec}(\mathcal{Z}, \mathbf{A}, \pi) = \textsc{Exec}(\mathcal{Z}, S, F)$$

For realistic cryptographic protocols, however, perfect emulation is impossible. Their security almost invariably relies

on computational hardness assumptions, so an unbounded attacker or environment can easily glean information beyond the ideal functionality's "acceptable" leakage. As a result, the random variables will not be identically distributed.

To still recover a meaningful notion of security, we include a security parameter n and compare the behaviors asymptotically in n. That is, instead of considering $\text{EXEC}(\mathcal{Z}, \mathbf{A}, \pi)$ to be a single random variable, we consider it to be a family of random variables, $\{\text{EXEC}_n(\mathcal{Z}, \mathbf{A}, \pi)\}$, one for each value of n. Two such families are *indistinguishable* if the difference between them shrinks very rapidly (usually exponentially) as n grows. This notion is defined formally as follows.

Definition 2 (Indistinguishability [16])**.** Two ensembles of binary probability distributions $X = \{X_n\}$ and $Y = \{Y_n\}$ are *indistinguishable*, denoted $X \approx Y$, if, for all $c \in \mathbb{N}$, there is some $N \in \mathbb{N}$ such that

$$\forall n > N. \, |\Pr[X_n = 1] - \Pr[Y_n = 1]| < n^{-c}$$

Indistinguishability is not enough by itself. The computational hardness assumptions of the cryptographic primitives force us to limit the computational power of the parties within the system. In particular, the UC framework demands that all protocols, attackers, and environments execute in polynomial time. The result is the following formal definition of computational UC security.

Definition 3 (Computational UC Emulation [16])**.** A poly-time protocol π *computationally* UC*-emulates* a functionality F, denoted $\pi \vdash_{\text{UC}}^{\approx} F$ if, for all PPT adversaries $\mathbf{A}$, there is a PPT simulator S, such that for all PPT environments $\mathcal{Z}$,

$$\text{EXEC}(\mathcal{Z}, \mathbf{A}, \pi) \approx \text{EXEC}(\mathcal{Z}, S, F)$$

B. Robust Compilation

The Robust Compilation (RC) framework [2, 3] formalizes security of compilers as their ability to preserve arbitrary classes of (hyper)properties [21] *robustly*, i.e., in the presence of active adversaries linked with compiled code.

Compilers (denoted by $[\![\cdot]\!]$) are functions that translate components, or *partial programs* (P), from a source language (S) to a target language ($\mathbf{T}$). In both languages, a partial program P can link with a *program context* A to form a *whole program*, denoted $A \bowtie P$. Whole programs come equipped with an operational semantics called a *robust trace semantics* ($\rightsquigarrow$), which captures all security-relevant behavior of $A \bowtie P$ in a trace of events ($\bar{t}$) where each event indicates whether it comes from the context or the program.

In their original work, Abate et al. [2] define what criteria compilers must meet to prove that they preserve:

- trace properties, including all trace properties, safety properties, and dense properties, a variation of liveness,
- hyperproperties, including all hyperproperties, subset-closed hyperproperties, and 2-hypersafety, and
- relational hyperproperties.

A compiler that robustly preserves all hyperproperties satisfies RHC, defined formally as follows.

Definition 4 (Robust Hyperproperty-Preserving Compiler)**.**

$$\vdash [\![\cdot]\!] : RHC \stackrel{\text{def}}{=} \forall P, \mathbf{A}. \exists A. \forall \bar{t}. \, \mathbf{A} \bowtie [\![P]\!] \rightsquigarrow \bar{t} \text{ iff } A \bowtie P \rightsquigarrow \bar{t}$$

For a compiler to preserve arbitrary hyperproperties, the traces exhibited by any compiled component while interacting with an arbitrary target context must be the same to the traces exhibited by a the source component linked with an existentially quantified source context.

C. Existing UC–RC Connection

As Patrignani et al. [43] prove, Definition 1 and Definition 4 are equivalent. By inspecting the quantifiers, and their order (which match), it is possible to see that UC functionalities correspond to RC source components, UC protocols correspond to RC compiled components, and UC environments correspond to RC traces. The elements that provide a central role in the security argument are also related. The universally quantified UC protocol attackers are still universally quantified RC target contexts and the existentially quantified UC simulators are still existentially quantified RC source contexts.

With this equivalence, Patrignani et al. [43] demonstrate that by providing an RHC proof for a compiler that translates a program encoding a functionality into its protocol, one can obtain a UC proof. More importantly, since Definition 4 amounts to proving trace equivalence between programs, Patrignani et al. [43] use the Deepsec tool [20] in order to provide the RHC proof and thus the first mechanized proof of UC.

However, the equivalence formalized by Patrignani et al. [43] only amounts to a perfect notion of UC. In the following section, we set out to lift this limitation.

III. Computational Robust Hyperproperty Preservation

To see the connection between perfect UC and RHC, we look at the relationship between the definition of UC security and the behavior of a program W, denoted $Behav(W)$. Patrignani et al. [43] define $Behav(W) = \{\bar{t} \mid W \rightsquigarrow \bar{t}\}$ as the set of traces W can produce, and a trace $\bar{t}$ as a pair $(\overline{\tau}, \rho)$, where $\overline{\tau}$ is a (potentially infinite) sequence of actions and ρ is the probability that W will produce $\overline{\tau}$. They also include all actions of the environment in $\overline{\tau}$ (including outputting the final bit), so we can view $Behav(W)$ as containing different probability distributions for different environments. If we let $Behav(W)|_{\mathcal{Z}}$ denote the probability distribution over final bits for just the subset of behaviors consistent with environment $\mathcal{Z}$, the structure of RHC begins to look very much like the structure of UC security.

Indeed, we can rephrase Definition 4 to require equality of behaviors and then universally quantify over environments. The result is the following definition of RHC.

$$\forall P, \mathbf{A}. \exists A. \forall \mathcal{Z}. \, \mathbf{Behav}(\mathbf{A} \bowtie [\![P]\!])|_{\mathcal{Z}} = Behav(A \bowtie P)|_{\mathcal{Z}}$$

And recall from Definition 1 that π perfectly UC-emulates F if

$$\forall \mathbf{A}. \exists S. \forall \mathcal{Z}. \, \text{EXEC}(\mathcal{Z}, \mathbf{A}, \pi) = \text{EXEC}(\mathcal{Z}, S, F)$$

These definitions have a nearly-identical structure, which we use as a guide for moving to computational security. There are two main differences between perfect UC and computational UC that we need to represent in the language of robust compilation. First, the equality on distributions is replaced with computational indistinguishability. Second, the relationship between real and ideal protocols (and thus target and source programs, respectively) is further loosened to consider only attackers, simulators, and programs that are *polynomial time*.

This section now presents these changes to the existing RC theory (Section III-A) followed by the novel notion of computational robust compilation ($CRHC$, Section III-B). Then, it presents what class of hyperproperties $CRHC$ preserves (Section III-C) before presenting the main result of this paper, connecting omputational UC and $CRHC$ (Section III-D).

A. Computational Indistinguishability

Defining indistinguishability of program behaviors requires modifying the structures introduced by Patrignani et al. [43]. Recall from Section II-A that indistinguishability, denoted $X \approx Y$, is defined over *families* of distributions, $X = \{X_n\}$ and $Y = \{Y_n\}$, and requires the ability of an adversary to separate X_n from Y_n to shrink rapidly as n grows. To modify the above definition of RHC to match the structure of computational UC security, we would like to say $Behav(W_1)|_{\mathcal{Z}} \approx Behav(W_2)|_{\mathcal{Z}}$.

Unfortunately, with $Behav(W)$ structured as a set of pairs $(\overline{\tau}, \rho)$, as described above, we can interpret $Behav(W)|_{\mathcal{Z}}$ as a single probability distribution, but not the requisite family of them. Luckily, because the semantics of the program are left abstract, we can straightforwardly introduce a security parameter n. We thus define a program execution by a triple $(\overline{\tau}, \rho, n)$ instead of a pair. Here $\overline{\tau}$ and ρ are the same as before, and n is a security parameter. With this change, we can view $Behav(W)$ as a *family of behaviors*, indexed by n, which we can similarly restrict by an environment to get a family of distributions. That is,

$$Behav_n(W) \stackrel{\text{def}}{=} \{(\overline{\tau}, \rho) \mid W \rightsquigarrow (\overline{\tau}, \rho, n)\}$$
$$Behav(W) \stackrel{\text{def}}{=} \{Behav_n(W)\}$$
$$Behav(W)|_{\mathcal{Z}} \stackrel{\text{def}}{=} \{Behav_n(W)|_{\mathcal{Z}}\}$$

Definition 2 now applies with these families of distributions.

Quantifying over all possible environments, however, results in a definition of statistical indistinguishability, not computational indistinguishability. Achieving the computational goal requires restricting to only polynomial-time environments. Because we assume the program interfaces of the different languages match, the space of environments is the same, we use the same PPT set to represent all probabilistic poly-time environments.

These two modifications are sufficient to define computationally indistinguishable program behaviors.

Definition 5 (Computational Indistinguishability of Programs)**.** Whole programs W_1 and W_2 have *computationally indistinguishable behavior* if

$$\forall \mathcal{Z} \in \text{PPT}.\ Behav(W_1)|_{\mathcal{Z}} \approx Behav(W_2)|_{\mathcal{Z}}$$

We denote this equivalence by $\approxeq$ for both whole programs and sets of traces. That is, the above equivalence defines both $W_1 \approxeq W_2$ and $Behav(W_1) \approxeq Behav(W_2)$.

B. Computational Robust Compilation

Recall that computational UC security allows not only this looser computational notion of indistinguishability, but also restricts all protocols, attackers, and simulators to be polynomial time. The language of robust compilation, however, quantifies over *all* programs and contexts, not just poly-time ones.

Without bounding the computational power of programs and contexts, we cannot hope to represent computational UC security. To see why, consider the single-bit commitment protocol due to Canetti and Fischlin [17] discussed in Section I. In the RC context, if we consider a compiler that takes Canetti and Fischlin's commitment ideal functionality and produces their commitment protocol, we would hope to call that compiler robust. But because the language of RC always considers *all* contexts (attackers) and a computationally unbounded attacker can break the protocol, we cannot. To address this concern, we expand the framework with a notion of *computationally-robust* preservation of hyperproperties that considers only polynomial-time programs and contexts.

As in UC, execution complexity depends on the context (adversary), the program (protocol), and how they interact. We therefore define a poly-time predicate over whole programs. As with the program semantics, we leave this predicate abstract to avoid the need to restrict to a specific computational model with specific execution times. Because the source and target languages may have different semantics and computational models, we include both a source-language predicate $poly$ and a target-language predicate $\mathbf{poly}$.

These predicates are sufficient to define a more permissive notion of robust compilation: the *computationally-robust hyperproperty-preserving compiler* ($CRHC$).

Definition 6 (Computationally-Robust Hyperproperty-Preservating Compiler)**.**

$$\vdash [\![\cdot]\!] : CRHC \stackrel{\text{def}}{=} \forall P.\, \forall \mathbf{A}.\, \mathbf{poly}(\mathbf{A} \bowtie [\![P]\!]) \implies \exists A.\, poly(A \bowtie P) \wedge (\mathbf{A} \bowtie [\![P]\!] \approxeq A \bowtie P)$$

This computational notion considers only executions of poly-time programs in both the source and target language. It demands that, for any target context $\mathbf{A}$ where $\mathbf{A} \bowtie [\![P]\!]$ is poly-time, there must be a source context A such that $A \bowtie P$ is also poly-time and the behaviors of the whole programs are computationally indistinguishable.

CRHC and Optimizations: $CRHC$ demands that, if there is *any* target context $\mathbf{A}$ such that $\mathbf{A} \bowtie [\![P]\!]$ is poly-time, then there must be some source context A such that $A \bowtie P$ is also poly-time. This requirement has an odd ramification:

a $CRHC$ compiler may not optimize a super-polynomial program into a poly-time one, say, by introducing memoization. This may appear to be a limitation, but from a security standpoint it is not. Computational UC security considers only the behavior of poly-time programs and poly-time ideal functionalities. It says nothing about the security of a system with super-polynomial protocols, adversaries, environments, or ideal functionalities. Correspondingly, the security defined by $CRHC$ does not aim to be meaningful for super-polynomial programs. Instead, it demands that the compiler produce super-polynomial outputs on any super-polynomial inputs, allowing us to restrict all relevant analysis to the polynomial case.

C. Identifying Preserved Hyperproperties

In the original work defining RHC, Abate et al. [2] identify that RHC is equivalent to a compiler robustly preserving all hyperproperties. Recall that a hyperproperty is a set of sets of traces. This means that, for $[\![\cdot]\!]$ to be RHC, then for any hyperproperty H and partial program P, if the behavior of P is in H for every possible source context, then the behavior of $[\![P]\!]$ is in H for every possible target context.

More formally, let $\mathcal{T} = \{\bar{t}\}$ be the set of all traces and $\mathbb{T} = \mathcal{P}(\mathcal{T})$ be the set of sets of traces. Then

$$\begin{aligned}\vdash [\![\cdot]\!] : RHC \iff \\ \forall H \subseteq \mathbb{T}.\, \forall P.\, (\forall A.\, Behav(A \bowtie P) \in H) \implies \\ (\forall \mathbf{A}.\, \mathbf{Behav}(\mathbf{A} \bowtie [\![P]\!]) \in H)\end{aligned}$$

This connection with RHC raises a question in our context: what hyperproperties does computational robustness preserve? Or, more generally speaking: which class of (hyper)properties can UC security actually express? One might guess that $CRHC$ is equivalent to computationally-preserving all hyperproperties. That is, preserving all hyperproperties when considering only poly-time programs. However, restricting to poly-time environments, which attempt to distinguish program behaviors, makes this guess incorrect. Instead, $CRHC$ corresponds to preserving a narrower class of hyperproperties, CH, which represent computationally-indistinguishable programs.

To define CH, we first define $CH(F)$, a single hyperproperty representing the family of behaviors that are computationally indistinguishable from what F can produce when given an appropriate context. Formally,

$$\begin{aligned}CH(F) \stackrel{\text{def}}{=} \{T \in \mathbb{T} \mid \exists S.\, poly(S \bowtie F) \\ \wedge\, (\forall \mathcal{Z} \in \text{PPT}.\, T|_{\mathcal{Z}} \approx Behav(S \bowtie F)|_{\mathcal{Z}})\}\end{aligned}$$

Intuitively, $Behav(W) \in CH(F)$ means that there is some simulator S such that the behavior of $S \bowtie F$ cannot be distinguished from the behavior of W. If we instead consider a partial program P and quantify over poly-time contexts,

$$\forall A.\, poly(A \bowtie P) \implies Behav(A \bowtie P) \in CH(F)$$

means that P computationally emulates F.

The class CH that a $CRHC$ compiler preserves (for poly-time programs) is precisely the set of all of these hyperproperties. That is, we define $CH \stackrel{\text{def}}{=} \{H \mid \exists F.\, H = CH(F)\}$ and obtain the following result.

Theorem 1 ($CRHC$ is Computational Preservation of CH)**.**

$$\begin{aligned}\vdash [\![\cdot]\!] : CRHC \iff \forall H \in CH.\\ \forall P.\, (\forall A.\, poly(A \bowtie P) \implies Behav(A \bowtie P) \in H) \implies \\ (\forall \mathbf{A}.\, \mathbf{poly}(\mathbf{A} \bowtie [\![P]\!]) \implies \mathbf{Behav}(\mathbf{A} \bowtie [\![P]\!]) \in H)\end{aligned}$$

This theorem, proven in Isabelle/HOL (`compRHCisRHPX`), is a special case of Theorem 3 (see Section IV).

Theorem 1 shows that $CRHC$ is equivalent to preserving secure emulation. That is, a compiler $[\![\cdot]\!]$ satisfies $CRHC$ if and only if, whenever a source program P computationally emulates some functionality F in the source language, then the compiled $[\![P]\!]$ also computationally emulates F in the target language.

D. Connecting Computational UC and CRHC

This equivalence between $CRHC$ and secure emulation of functionalities suggests a similarly deep connection to computational UC security. Indeed, the modifications to get from RHC to $CRHC$ were designed explicitly to mirror the differences between perfect UC and computational UC. To prove this correspondence formally, we rely on four axioms laid out by Patrignani et al. [43] to move between the Interactive Turing Machine (ITM) semantics of the UC framework and the abstract semantics of the RC framework. We modify Axioms 1, 2, and 4 only to add the security parameter n.

The first uses the function $z(\overline{m})$ to define a *canonical environment* for a trace prefix $\overline{m}$ that produces exactly this prefix and then halts the execution with final bit 1. It says that a trace prefix is possible if and only if a corresponding execution is possible for the ITM.

Axiom 1 (UC and RC Semantics [43])**.** *If $\overline{m} = (\overline{\mu}, \rho, n)$ and $\overline{\mu}$ is a finite sequence of actions, then*

$$A \bowtie P \rightsquigarrow \overline{m} \iff \Pr[\text{EXEC}_n(z(\overline{m}), A, P) = \overline{\mu}] = \rho > 0.$$

The second axiom requires that the canonical environment for a trace correctly represent the behavior of all environments that produce the same trace.

Axiom 2 (Canonical Environment Correctness [43])**.** *If $\mathcal{Z}$ is non-probabilistic, $\overline{m} = (\overline{\mu}, \rho, n)$, and for some $\mathbf{A}'$ and π', $\Pr[\text{EXEC}_n(\mathcal{Z}, \mathbf{A}', \pi') = \overline{\mu}] > 0$, then for all $\mathbf{A}$ and π,*

$$\Pr[\text{EXEC}_n(\mathcal{Z}, \mathbf{A}, \pi) = \overline{\mu}] = \Pr[\text{EXEC}_n(z(\overline{m}), \mathbf{A}, \pi) = \overline{\mu}]$$

This axiom assumed that the environment $\mathcal{Z}$ is non-probabilistic, meaning the probability of $\overline{m}$ depends only on the randomness of $\mathbf{A}$ and π. This assumption simplifies reasoning and is not a limitation, because we also assume that anything a probabilistic environment can do, a non-probabilistic one can do as well.

Axiom 3 (Non-Probabilistic Environment Completeness)**.** *For any $\mathcal{Z} \in$ PPT, if $\text{EXEC}(\mathcal{Z}, \mathbf{A}, \pi) \not\approx \text{EXEC}(\mathcal{Z}, S, F)$, then there exists some non-probabilistic poly-time $\mathcal{Z}'$ such that $\text{EXEC}(\mathcal{Z}', \mathbf{A}, \pi) \not\approx \text{EXEC}(\mathcal{Z}', S, F)$.*

Patrignani et al.'s [43] version of Axiom 3 uses (in)equality instead of (in)distinguishability and does not bound $\mathcal{Z}$ and $\mathcal{Z}'$,

but is otherwise the same. Intuitively, this axiom is valid because, for any distinguishing $\mathcal{Z}$, one can select from the random choices made by $\mathcal{Z}$ (but not the attacker or protocol) the ones that maximize $\mathcal{Z}$'s ability to distinguish the real and ideal worlds. Fixing those choices produces a deterministic environment that distinguishes at least as well as $\mathcal{Z}$.

Finally, we assume that trace prefixes specify whether or not the environment has decided on a final bit b, and if so, what that value is. The UC framework assumes the environment terminates execution with that final bit, but we follow Patrignani et al. [43] and abstract this process over an arbitrary encoding with an extraction function $\beta : \overline{\mu} \to \{0, 1, \bot\}$ with the following property.

Axiom 4 (Finite Traces Contain the Final Bit [43])**.**

$$\Pr[\text{EXEC}_n(\mathcal{Z}, \mathbf{A}, \pi) = b] = \sum_{\beta(\overline{\mu})=b} \Pr[\text{EXECT}_n(\mathcal{Z}, \mathbf{A}, \pi) = \overline{\mu}]$$

Using these axioms, we are able to prove the desired correspondence between UC security and $CRHC$.

Theorem 2 (Computational UC and $CRHC$ Coincide)**.**

$$\vdash [\![\cdot]\!] : CRHC \iff \forall P. [\![P]\!] \vdash_{\text{UC}}^{\approx} P$$

This theorem is verified in Isabelle/HOL (`compRHCeqUC`).

As we will see in Section VI, Theorem 2 creates a powerful new means of mechanizing a proof of UC. It is now sufficient to show that the compiler that translates F into π satisfies $CRHC$ or, equivalently, preserves CH for polynomial-time programs. That is, it suffices to prove computational indistinguishability between F and π, given the correct simulator S. These proofs can be carried out using existing tools. We will use CRYPTOVERIF [13] and discuss alternatives in Section VII.

IV. GENERALIZING EQUIVALENCES AND PREDICATES

In Sections III-A and III-B we modified RHC to obtain the computational analogue $CRHC$. Very little about that process, however, was specific to polynomial time or computational indistinguishability. Indeed, we can generalize nearly all of those modifications and achieve a far more general result that subsumes our results from Section III, as well as the perfect security results of Patrignani et al. [43], and points to other notions of security as well.

In Section III-A, we loosened the requirement of *equality* of behaviors to *computational indistinguishability*. This change required modifying the definition of traces to interpret a set of traces as a family of probability distributions. To capture both the original equality view and this looser view, we can generalize to an arbitrary equivalence relation $\equiv$ over sets of traces, which we will apply to program behaviors.

In Section III-B, we expanded the robust compilation framework with polynomial-time predicates. We required that behaviors only be preserved for poly-time programs, though we restricted the existentially quantified source contexts to be poly-time as well. There is nothing special about poly-time in that process. Indeed, we could instead have used abstract predicates Q and $\mathbf{Q}$ over source and target programs.

Making this change leads to a more general notion of *predicate-robust hyperproperty preservation* ($PRHC$) parameterized on an equivalence and two predicates.

Definition 7 (Predicate-Robust Hyperproperty-Preservating Compiler)**.**

$$\vdash [\![\cdot]\!] : PRHC(\equiv, \mathbf{Q}, Q) \stackrel{\text{def}}{=} \forall P. \forall \mathbf{A}. \mathbf{Q}(\mathbf{A} \bowtie [\![P]\!]) \implies \exists A. Q(A \bowtie P) \wedge (\mathbf{Behav}(\mathbf{A} \bowtie [\![P]\!]) \equiv Behav(A \bowtie P))$$

That is, source contexts linked with source programs can produce all the behavior of compiled programs linked with target contexts (up to $\equiv$) when only considering programs satisfying Q and $\mathbf{Q}$, the respective predicates for the source and target language.

This notion generalizes both RHC and $CRHC$. If we instantiate $\equiv$ with set equality ($=$) and both Q and $\mathbf{Q}$ with the trivial predicate True that holds for all programs, $PRHC(=, \text{True}, \text{True})$ requires source contexts to produce exactly the behavior of target contexts when considering all programs—precisely RHC.

Proposition 1 ($PRHC$ expresses RHC)**.**

$$\vdash [\![\cdot]\!] : PRHC(=, \text{True}, \text{True}) \Updownarrow \forall P. \forall \mathbf{A}. \exists A. \mathbf{Behav}(\mathbf{A} \bowtie [\![P]\!]) = Behav(A \bowtie P)$$

which is the definition of $\vdash [\![\cdot]\!] : RHC$.

Similarly, we can instantiate $\equiv$ with computational indistinguishability ($\approx$), interpreting traces as families of distributions and restricting to poly-time environments, Q with $poly$, and $\mathbf{Q}$ with $\mathbf{poly}$. Here $PRHC(\approx, poly, \mathbf{poly})$ says that source contexts must produce behavior indistinguishable from target contexts when considering only polynomial programs—precisely $CRHC$.

Proposition 2 ($PRHC$ expresses $CRHC$)**.**

$$\vdash [\![\cdot]\!] : PRHC(\approx, \mathbf{poly}, poly) \iff \vdash [\![\cdot]\!] : CRHC$$

We also generalize the result from Section III-C identifying which hyperproperties $CRHC$ compilers preserve. To formalize this idea, we extend robust preservation of a specific class X of hyperproperties, due to Abate et al. [2], with predicates in the same way that $PRHC$ extends RHC, producing the following definition of *predicate-robust hyperproperty preservation* ($PRHP$).

Definition 8 (Predicate-Robust Hyperproperty Preservation)**.**

$$\vdash [\![\cdot]\!] : PRHP(X, \mathbf{Q}, Q) \stackrel{\text{def}}{=} \forall H \in X. \forall P. (\forall A. Q(A \bowtie P) \implies Behav(A \bowtie P) \in H) \implies (\forall \mathbf{A}. \mathbf{Q}(\mathbf{A} \bowtie [\![P]\!]) \implies \mathbf{Behav}(\mathbf{A} \bowtie [\![P]\!]) \in H)$$

That is, a $PRHP(X, \mathbf{Q}, Q)$ compiler is one where, for any hyperproperty $H \in X$, if a source program P produces

only behaviors in H when restricting to source contexts that satisfy Q, then the compiled program $[\![P]\!]$ must also produce only behaviors in H when considering only target contexts satisfying $\mathbf{Q}$.

As with $PRHC$, this generalizes previous notions. It can express robust preservation of all hyperproperties through $PRHP(\mathcal{P}(\mathbb{T}), \mathsf{True}, \mathsf{True})$, setting X to all hyperproperties and Q and $\mathbf{Q}$ to True to consider all contexts. It can also express computationally-robust preservation of CH (Section III-C), by setting $X = CH$ and Q and $\mathbf{Q}$ to $\mathbf{poly}$ and $poly$, respectively.

Finally, we can relate $PRHC$ and $PRHP$ by defining the class of hyperproperties that a $PRHC$ compiler preserves in terms of the equivalence $\equiv$ and the predicates Q and $\mathbf{Q}$. As in Section III-C, we define the class of hyperproperties by the set of functionalities F that a program securely emulates. Where $CH(F)$ is the set of behaviors computationally indistinguishable from F, the general hyperproperty $Hyp_{\equiv,Q}(F)$ is the set of behaviors equivalent to F up to $\equiv$ when considering only whole programs that satisfy predicate Q. Formally,

$$Hyp_{\equiv,Q}(F) \stackrel{\text{def}}{=} \{T \in \mathbb{T} \mid \exists S.\, Q(S \bowtie F) \wedge T \equiv Behav(S \bowtie F)\}$$

As with CH, we again define a class of hyperproperties as the set of all of these hyperproperties:

$$HypCls_{\equiv,Q} \stackrel{\text{def}}{=} \{H \mid \exists F.\, H = Hyp_{\equiv,Q}(F)\}.$$

This is precisely the class we are looking for. When using the same equivalence and the source-language predicate Q, predicate-robustly preserving this class is equivalent to predicate-robustly preserving behavior up to $\equiv$.

Theorem 3 ($PRHC$ is $PRHP$)**.**

$$\vdash [\![\cdot]\!] : PRHC(\equiv, \mathbf{Q}, Q) \Updownarrow \vdash [\![\cdot]\!] : PRHP(HypCls_{\equiv,Q}, \mathbf{Q}, Q)$$

This theorem is verified in Isabelle/HOL (`RHPXeqRRHC`).

Note that, for all F, $CH(F) = Hyp_{\cong,poly}(F)$, meaning $CH = HypCls_{\cong,poly}$. Theorem 1 therefore follows as a special case of Theorem 3 using Proposition 2.

A. Connecting to UC Security

One obvious question this generalization raises is: how do these generalized notions of RC correspond to UC security? Recall that the definitional structure of UC security is extremely similar to the structure of RHC (and $CRHC$ and $PRHC$). UC security explicit demands environments that distinguish between executions while $PRHC$ allows arbitrary equivalence relations over behaviors, but otherwise they are nearly identical. Because the equivalence relation is arbitrary, we can consider the class of equivalences that consider differences in behaviors between the source and target program for each environment separately, as we did in Section III.

Taking this view, the predicates Q and $\mathbf{Q}$ represent restrictions on the behavior of the protocols, ideal functionalities, attackers, and simulators, and the equivalence $\equiv$ must specify any restrictions on environments as well as how similar the behaviors must be. Setting the predicates to True and $\equiv$ to $=$, as we did above, yields that requirement that behaviors must be identical, with no restriction on protocols, functionalities, attackers, simulators, or environments. That definition corresponds precisely to perfect UC security (Definition 1). Since we have already shown the same assignments produce the definition of RHC, this correspondence immediately re-proves the result of Patrignani et al. [43].

Similarly, if we use $poly$ and $\mathbf{poly}$ as predicates and computational indistinguishability, we recover a definition of computational UC security (Definition 3), and immediately recover Theorem 2.

We are not, however, limited to these two cases. For instance, consider using the trivial predicate True to leave protocols and attackers unrestricted, and instantiating $\equiv$ with $\cong$, which leaves environments unrestricted but demands only *indistinguishable* behaviors. That is,

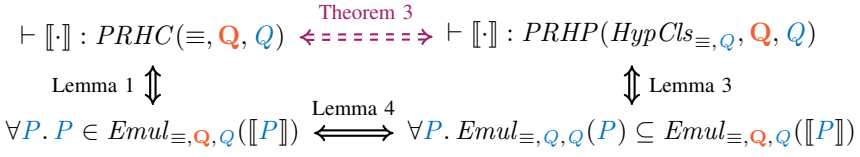
$$T_1 \cong T_2 \stackrel{\text{def}}{=} \forall \mathcal{Z}.\, T_1|_{\mathcal{Z}} \approx T_2|_{\mathcal{Z}}$$

The result is not perfect UC security, as the behaviors may differ, but it is also not computational, as there is no restriction on the complexity of any component of the system. Instead, $PRHC(\cong, \mathsf{True}, \mathsf{True})$ corresponds to a third version of UC security: *statistical* UC security, where the distributions of behaviors must be statistically close, but not identical.

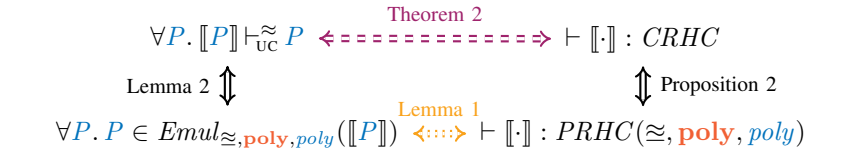
There are also other indistinguishability notions that appear in both the cryptography and language-based security literatures, and this result can connect them to each other. Examples from the cryptographic domain include Rényi divergence [9], which is often used in lattice-based cryptography as it provides a definition for when a search problem is computationally difficult, Kullback-Leibler divergence [39], which has been used to to simplify proofs relating to fundamental definitions like adversarial advantage, and more [e.g. 5].

In language-based security, properties like noninterference [28] and observational determinism [38, 47] are often defined using an equivalence that erases certain (secret) parts of the trace and demands the remaining (public) portion be identical [21, 48, 55]. Different variations of these equivalences define different notions of noninterference (e.g., termination-sensitive or insensitive, timing sensitive, or constant time), or generalize to programs that include explicit declassifications or endorsements [19, 49, 54]. Even other equivalence notions, like differential privacy [22], may be possible, particularly in combination with existing language-based results [51, 52, 57].

By inserting these equivalences into our framework, we immediately obtain a formal definition for the security they define from both a robust compilation and a UC standpoint. The result is a meaningful way to relate more cryptographic security to secure compilation and more language-based security to cryptography. Moreover, it provides the language needed to combine the two. For example, some existing work



(a) Proof structure of Theorem 3 using arbitrary $\equiv$, $\mathbf{Q}$, and Q.



(b) Proof structure of Theorem 2.

Fig. 1. Visual depiction of the proofs of our main theorems. We prove solid black implications directly, derive dotted the orange implication as a special case, and conclude dashed purple results.

combines noninterference-style definitions with computational hardness [25, 26], and our work gives a way to situate those equivalences more broadly points to ways to extend and expand them.

Finally, the relation $\equiv$ does not need to be an equivalence relation. Our proofs demand only that it is reflexive and transitive (i.e., a preorder), *not* symmetric. Indeed, instantiating $\equiv$ with $\subseteq$ produces a notion of behavioral refinement.

V. Proof Approach

To prove Theorems 1 and 3, we develop an approach that connects directly to the definition of UC security, and allows a simple and direct proof of Theorem 2. Extending it with the generalization in Section IV provides similar proof connecting RHC to perfect UC security, simplifying and clarifying the proof by contradiction of Patrignani et al. [43].

Our proofs rely on the idea of an *emulation set*, which we denote $Emul(P)$, that represents the set of functionalities that P securely emulates. As in Section IV, we use a completely general definition of "securely emulates" and parameterize $Emul$ on an arbitrary equivalence $\equiv$ and predicates Q and Q'.

$$\begin{aligned} &Emul_{\equiv,Q,Q'}(P) \stackrel{\text{def}}{=} \\ &\quad \{F \mid \forall A.\ Q(A \bowtie P) \implies \exists S.\ Q'(S \bowtie F) \\ &\qquad\qquad \wedge\ Behav(A \bowtie P) \equiv Behav(S \bowtie F)\}. \end{aligned}$$

This definition follows a very similar structure to $PRHC$. Setting $\equiv$ to set equality ($=$) and both predicates to True again yields a definition of perfect emulation, while setting $\equiv$ to computational indistinguishability ($\approx$) and the predicates to $poly$ produces a definition of computational emulation. The structure also allows for a very simple proof that a compiler is $PRHC$ if and only if all compiled programs securely emulate their source program (using the same predicates and equivalence).

Lemma 1 (Emulation is $PRHC$)**.**

$$\vdash [\![\cdot]\!] : PRHC(\equiv, \mathbf{Q}, Q) \iff \forall P.\ P \in Emul_{\equiv,\mathbf{Q},Q}([\![P]\!])$$

Moreover, Axioms 1 – 4 (Section III-D) directly connect this definition to UC-style semantics. For instance, the following results hold.

Lemma 2 (Emulation is UC)**.**

$$\begin{aligned} \pi \vdash^{=}_{\text{UC}} F &\iff F \in Emul_{=,\mathsf{True},\mathsf{True}}(\pi) \\ \pi \vdash^{\approx}_{\text{UC}} F &\iff F \in Emul_{\approx,\mathbf{poly},poly}(\pi) \end{aligned}$$

Lemmas 1 and 2 combine to provide direct proofs for the perfect security result of Patrignani et al. [43] and Theorem 2.

Emulation sets are also helpful in stating the class of hyperproperties a $PRHC$ compiler preserves, making them useful in the proof of Theorem 3. In particular, a program P's behavior is in $Hyp_{\equiv,Q'}(F)$ in any context satisfying predicate Q precisely when $F \in Emul_{\equiv,Q,Q'}(P)$. That is,

$$\begin{gathered} \forall \mathbf{A}.\ \mathbf{Q}(\mathbf{A} \bowtie \mathbf{P}) \implies \mathbf{Behav}(\mathbf{A} \bowtie \mathbf{P}) \in Hyp_{\equiv,Q}(F) \\ \Updownarrow \\ F \in Emul_{\equiv,\mathbf{Q},Q}(\mathbf{P}) \end{gathered}$$

Applying this insight to $PRHP$ (Definition 8) yields the following result.

Lemma 3 (Emulation specifies $PRHP$)**.**

$$\begin{gathered} \vdash [\![\cdot]\!] : PRHP(HypCls_{\equiv,Q}, \mathbf{Q}, Q) \\ \Updownarrow \\ \forall P.\ Emul_{\equiv,Q,Q}(P) \subseteq Emul_{\equiv,\mathbf{Q},Q}([\![P]\!]) \end{gathered}$$

The final hurdle to proving Theorem 3 is to notice that this subset relationship is equivalent to containment.

Lemma 4 (Emulation Subset is Containment)**.**

$$F \in Emul_{\equiv,\mathbf{Q},Q}(\mathbf{P}) \iff Emul_{\equiv,Q,Q}(F) \subseteq Emul_{\equiv,\mathbf{Q},Q}(\mathbf{P})$$

This result relies on the reflexivity of $\equiv$ to show that $F \in Emul_{\equiv,Q,Q}(F)$, and the transitivity of $\equiv$ to show that anything that F emulates $\mathbf{P}$ also emulates. Notably, there is no need for $\equiv$ to be symmetric.

The proofs of all theorems in Sections III and IV, which are verified in Isabelle/HOL, follow directly from the combination of these four lemmas using the structures shown in Figure 1.

VI. A Mechanized proof of UC for Wireguard

We now describe how to leverage the results from Section III into a mechanized proof of UC security for the Wireguard protocol using the CryptoVerif tool. In particular, Theorem 2 means we can prove UC security by proving that a compiler that transforms the Wireguard ideal functionality into the Wireguard protocol satisfies $CRHC$. We encode our notion of whole programs as CryptoVerif games, and since CryptoVerif is designed to analyze computational equivalence $\approx$ between two games, we can use it to verify $CRHC$.

We begin with some background on CryptoVerif and Wireguard (Sections VI-A and VI-B) before describing how we model Wireguard in CryptoVerif (Section VI-C). Finally, we discuss the computational indistinguishability proof itself and limitations (Sections VI-D and VI-E).

A. The CryptoVerif Tool

CryptoVerif is a mechanized prover for properties of security protocols in the computational model. A CryptoVerif proof is a polynomial-length sequence of cryptographic games—where an active attacker attempts to break some security property and is allowed to run multiple sessions in parallel. Each game is represented in a language with a probabilistic semantics where all computations run in polynomial time, and the goal is to show that each game is indistinguishable from the next. CryptoVerif proofs are thus proofs of computational indistinguishability between two games, one of which is generally trivially secure (e.g., distinguishing between two encryptions of 0).

We use the game language of CryptoVerif as an instance for both the source and target languages of the compilers described in Section III. Those languages require semantics that are probabilistic and where processes run in polynomial time, assumptions which CryptoVerif satisfies. Our strategy is to encode both the ideal functionality and the protocol as games, and prove that they are computationally indistinguishable. This, in turn, shows that the compiler that translates the functionality into the protocol (and does nothing else) satisfies $CRHC$, and thus the protocol UC-realises the functionality.

CryptoVerif provides two input languages, channels and oracles. We use the oracle language, as it is closer to CryptoVerif's internal representation and thus grants more flexibility in the encoding. In this model, the parties to a protocol are represented by oracles, which are simple probabilistic programs with shared state. The adversary controls the network as well as the scheduling of messages by invoking these oracles. An adversary invoking an oracle models sending a message to a party and then receiving a response. This structure supports the attacker injecting messages—by invoking an oracle with an input it generated—dropping messages—by declining to send the output of one oracle as the input to another—or simply observing all network traffic. CryptoVerif models parties that answer many requests in a uniform way, such as servers, by replicating an oracle: given some finite bound n, the attacker can use n different copies of the oracle, each with their own state.

B. The Wireguard Protocol

Wireguard is a widely-used protocol for establishing a VPN tunnel between two remote hosts in order to securely encapsulate all Internet Protocol (IP) traffic between them. It consists of two parts: a key exchange and a record subprotocol. To establish a tunnel, the key exchange subprotocol requires the IP address and long-term public key for the remote host. It then uses an instantiation of the Noise framework for key exchange with fast, modern cryptographic primitives, like Curve25519 and BLAKE2, to establishes two ephemeral keys (one per direction) that parties use to encrypt and authenticated messages in the subsequent record subprotocol. The record subprotocol can then use these symmetric keys to construct a secure channel.

We verify a model of Wireguard introduced by Lipp et al. [37], which is already encoded in CryptoVerif.[1] For the sake of simplicity, we focus on the 2-party version of Wireguard, with a sender and receiver who are both honest throughout the run.

We also only model the record protocol, where payloads are transmitted after parties have agreed on a common, ephemeral symmetric session key. The compositional nature of UC security means that one could obtain a proof of the full Wireguard protocol simply by combining our proof with a corresponding proof for the key exchange protocol, which we leave for future work. We make this choice due to a conceptual challenge in proving the security of key exchange. The key confirmation message in Wireguard—as well as other secure channel protocols like TLS—leaks a very small amount of information about the key, complicating composition arguments. Fischlin et al. [24] discuss this problem and provide a (non-composable) solution, while the model of Lipp et al. [37, p. 242] omits this message altogether.

Finally, we introduce a modification to the original model. As an argument to the protocol, Lipp et al. [37] include a secret bit that the two parties share. Before sending a payload, a party waits for two messages from the adversary and uses the secret bit to determine which one to transmit. The attacker must then determine the value of this secret bit. This modeling choice deviates from any actual Wireguard implementation, which should not include this secret bit or a second payload. Instead, a more faithful model would take this direction from the environment, not the adversary. To represent this more realistic model, we build a simple model of communication between the Wireguard parties (sender and receiver), add a dummy adversary, and use the CryptoVerif attacker to model the environment. We then add functionality from Lipp et al.'s model in a piece-wise fashion until we have recovered the full record protocol.

[1] They provide several models. We build on a model in which long-term keys can be dynamically corrupted (`WG.25519.AB-BA.S_i_compr.S_r_compr.replay_prot.cv`).

C. Modeling Wireguard in CryptoVerif

Model Structure: Recall that CryptoVerif proves computational indistinguishability between two cryptographic games, one representing the "real world," which we denote $\mathbf{A_d} \bowtie \mathbf{P_{WG}}$, and one representing the "ideal world," which we denote $S_{WG} \bowtie P_{WG}$. The real world consists of the sender and the receiver (both in $\mathbf{P_{WG}}$) and the dummy attacker ($\mathbf{A_d}$). The ideal world consists of the sender and receiver's functionalities (P_{WG}) and a simulator (S_{WG}) that we devise to carry out the UC proof. Both the real and the ideal worlds interact with the same environment. In our model, the role of the environment is taken by the CryptoVerif attacker.

The Simulator: The simulator plays a key role in UC proofs, since it determines the success of the proof itself. Our simulator is inspired by simulators used in UC proofs of encryption functionalities. As the simulator performs multiple input/output steps, it is split over multiple oracles. The oracle below is the part of the simulator that receives a cyphertext from the environment and sends it to the functionality.

```
1 let Sim() =
2  foreach i_Nrr <= n_M do (
3   Oe2aR (xc:bitstring, xn:counter_t) :=
4     get rcvd_counters(xn) in yield
5     else insert rcvd_counters(xn);
6          get simtable(zm,tc,tn)
7          suchthat tc = xc && tn = xn
8          in run SenderFSim(xn)).
```

Line 2 creates n_M instances of the simulator, indexed by i_{Nrr}. Line 3 declares n_M many instances of the `Oe2aR` oracle (**O**racles modelling communication from the **e**nvironment via the dummy **a**ttacker to the **R**eceiver), which together constitute the simulator. Each `Oe2aR` instance receives a message containing a bitstring `xc` and a counter `xn` from the environment. Then it checks if the counter has been received in the past. If it has, the simulator aborts (`yield`), otherwise it registers the counter. Finally, each oracle instance checks if it has already received the bitstring with the same counter, and continues as `SenderFsim`, which forwards the bitstring to the sender part of the functionality.

Using CryptoVerif Oracles: We use the oracle interface of CryptoVerif to describe both games and the attacker. Concretely, each CryptoVerif game provides the attacker with a set of oracles that can be queried by the attacker. CryptoVerif does not provide private communication (e.g., private channels), and this affects both the protocols and the various parties encoded in the real and ideal games.

Modeling Private Communication: Private communication between protocol/functionality and attacker/simulator is necessary to ensure that the trace does not immediately reveal whether a run is in the real world or ideal world. In a process calculus, private channels can serve this purpose [12, 20]. In a stateful language, shared state is sufficient, but it requires proper encoding [10, 44]. CryptoVerif does not provide private communication natively, so we must encode it.

Oracles are allowed to share state and access each others variables directly. While unusual from a semantics perspective, this method is often preferred by cryptographers for its simplicity, so we rely on the exposure of oracle states to model private communication between them. If oracles are replicated (as in the case here), their variables must be accessed as arrays. For instance, `zm[simId]` would access the variable `zm` defined on line 6 of the copy of the above `Sim` oracle with ID `simId`. Concretely, this means *any* oracle with the following code snippet could access `zm`.

```
find simId <= n_M suchthat defined(zm[simId])
    then return(zm[simId]).
```

Here, the construct `find ... suchthat` non-deterministically sets such an index matching the specified requirements: `simId` $\leq$ `n_M`, and oracle `simId` has initialized `zm`.

In the real world, the only necessary private communication is between the protocol ($\mathbf{P_{WG}}$) and the dummy attacker ($\mathbf{A_d}$). Since the dummy attacker $\mathbf{A_d}$ is just a proxy that forwards all messages between $\mathbf{P_{WG}}$ and the environment, represented by the CryptoVerif attacker, we elide the dummy attacker and have $\mathbf{P_{WG}}$ communicate directly with the environment / CryptoVerif attacker. For example, the `Sender` directly returns to the CryptoVerif attacker in `Oe2S`, without calling the dummy attacker.

```
let Sender(ks:key_t) =
  foreach i_Nis <= n_M do (
    Oe2S(m:bitstring, counter:counter_t) :=
      let preparemsgsucc(cipher_data:bitstring)
          = prepare_msg(m, counter, ks)
      in return(cipher_data)
  ).
```

The ideal world requires more interesting private communication between the ideal functionality (P_{WG}) and the simulator (S_{WG}). However, S_{WG} is not stateless, so we need to manually inline the receiving party where it is called. We carefully track names to ensure that the receiving entity only accesses information that would be passed by communication. Unfortunately, this manual encoding can be error prone, since CryptoVerif allows the receiver to access any part of the sender state without a warning.

To illustrate this private communication, see the following `SenderFSim` oracle, which accesses the variables `m`, `counter`, and `c` that are defined outside of its scope, in the oracle `Sender` above.

```
let SenderFSim(xn:counter_t) =
  find senderid <= n_M
    suchthat defined(m[senderid],
                     counter[senderid],
                     c[senderid])
       && counter[senderid] = xn
  then return(m[senderid]).
```

`SenderFSim` constitutes the adversary/simulator interface of the sender part of the functionality (whereas `Sender` constitutes the environment interface and sets `m`). Because we model multiple sessions of the overall protocol running in parallel, the simulator may receive an out-of-order message from the environment, so the simulator needs to identify the correct session of the functionality. It does so by providing the

(public) counter `xn`, which `SenderFSim` uses to search the correct (internal) session identifier. First, the functionality finds the session, identified via `senderid`, that emitted the cyphertext. The functionality searches for a session that (a) is terminated, i.e., one where `m[senderid]`, `Fcounter[senderid]` and `c[senderid])` are defined, and (b) where the oracle's counter `Fcounter[senderid]` matches the counter `xn` received from the simulator. Up to now, we have simply encoded that the functionality, after producing its output to the simulator, waits for another message that it matches against `xn`; this is necessary because oracles can only produce one output per input. Finally, the functionality returns the message of the session now identified: `m[senderid]`.

The final model is around 200 lines of CRYPTOVERIF code.

D. Computational Indistinguishability Proof

We prove our two models (real and ideal) are computationally indistinguishable through a series of eight CRYPTOVERIF games that the tool does not find automatically. Two of those games are cryptographic reductions, while the others are perfect equivalences. We guide CRYPTOVERIF to these games with a proof script consisting of seven steps.

The scrip starts from the real world $\mathbf{A_d} \bowtie \mathbf{P_{WG}}$. The first step applies the IND-CTXT assumption of the encryption scheme, which guarantees ciphertext integrity. This check matches a structural check in the ideal functionality and ensures that every received ciphertext was sent by the sender. The second step applies the IND-CPA assumption of the encryption scheme, which guarantees ciphertext confidentiality. This step replaces message encryptions with encryptions of zero. The other five steps consist of removing dead code, inlining variables, and minimally modifying code structure to produce exactly the ideal world $S_{WG} \bowtie P_{WG}$.

E. Discussion

The lack of private communication such as private channels led us to use cross-oracle state sharing to transfer information between oracles and to inline parties where they are called. This is a manual process that is highly prone to encoding errors, and we believe adding private channels to CRYPTOVERIF would mitigate this issue. Indeed, this feature seems available internally, but we confirmed through the manual and communication with the maintainers that is is not exposed to the user. In the long term, the addition of a module system that allows for compositional proofs might be even better, as it could syntactically ensure that composition is correctly encoded, for example according to Patrignani et al. [43, Axiom 5 to 7]. In addition, such a system could easily validate that states are properly encapsulated, e.g., that simulator oracles cannot directly access the state of the functionality.

VII. RELATED WORK

The most closely-related work is that of Patrignani et al. [43], which highlights the UC–RC connection but does not broach the topic of computational indistinguishability. The authors use the connection to mechanize the proof that 1-bit commitment protocols [16, 36] are UC secure in the static as well as in the dynamic corruption cases.

A. Universally Composable Cryptographic Models

Universal composability [16] extended simulated-based notions of correctness and privacy for multi-party computation [29] to the interactive case, additionally ensuring that security guarantees are preserved when a protocol is used as a subroutine within a larger protocol. Many competing frameworks [7, 15, 30, 35, 53] followed to improve on perceived inadequacies in the model or to correct subtle errors in the proofs. Currently, the original UC framework is in its sixth version. A common point of contention is how to define polynomial runtime in a reactive system. Hofheinz et al. [31] explain the problem in great detail and propose a solution that is (conceptually) reflected in all these frameworks [15, 16, 30, 35, 53], at least in their latest versions. Our abstract poly-time predicate over attacker-protocol combinations (see Section III) can capture the poly-time notions in each of them.

These frameworks define UC bottom-up, fixing a plethora of technical details like the machine model (often interactive Turing Machines), message formats, message and process schedulers, virtualisation mechanisms, addressing mechanisms, etc. None of these details are at all similar to how networks and programs operate in the real world, yet many of the differences between the frameworks boil down to technical minutia at very low-level of abstraction. This is reflected in the proofs, which are as hard to formalize as they are to write down.

By contrast, our results operate at a much higher level of abstraction. Patrignani et al. [43] provide a high-level composition proof for this model, which we verified in Isabelle/HOL is not impacted by any of our modifications (`composition`). Indeed, that result relies on a small set of assumptions [43, Axioms 5 to 7] that most likely hold for any of the frameworks discussed above.

Canetti, Stoughton, and Varia [18] confirmed these issues, when they mechanized parts of UC in EasyCrypt [10], lamenting that "despite the relative simplicity [of their case studies], [proving UC] took an immense amount of work." It is thus unsurprising that other attempts to mechanize the concept [4, 11, 36] avoid this technical baggage and build their frameworks on formal languages that provide a much higher degree of abstraction. Still, all these frameworks perceive UC as a property between programs written in one and the same language. With Theorem 2, one can use RC as a tool to transfer such properties across language boundaries.

B. Alternative Tools for CRYPTOVERIF

We verified our case study in CRYPTOVERIF, but there are a variety of other tools for cryptographic verification.

EasyCrypt [23] is a stand-alone theorem prover with focus on cryptographic primitives and small protocols. It reasons at at a code level using probabilistic relational Hoare logic and external SMT solvers. There are also various embeddings of probabilistic languages in general-purpose theorem provers like CryptHOL [11] and FCF [44]. They all require most of

the proof to be written out, although some automated proof tactics can help with simple steps.

Squirrel [8] is a protocol-specific prover for computational indistinguishability based on a computationally-sound attacker model that can be symbolically reasoned about. In contrast to CRYPTOVERIF and EasyCrypt, which focus on program transformations to deduce equivalences, Squirrel's reasoning focusses on (symbolic) traces. Currently Squirrel proofs are as detailed as proofs in EasyCrypt and similar tools, though the project aims to reduce the complexity.

Only EasyCrypt has previously been used for UC-style proofs [18]. Leveraging our results, we were able to produce a proof in CRYPTOVERIF, which we chose because it promised a higher degree of automation. Our case study confirms this promise, though manual guidance was still required.

C. Language-Based Tools for Cryptographic Security

Several language-based tools exist with the goal of easing the process of developing and verifying secure cryptographic protocols. Languages like Wysteria [45], Wys* [46] and Symphony [50] are designed to simplify the design and implementation of secure distributed systems using cryptography. Viaduct [6] and Jif/Split [56, 58] use security policies specified by information flow labels to automatically partition programs, synthesizing uses of particular cryptographic primitives when necessary. These systems generally do not contain formal proofs of security, and we hope our results will make it easier to provide them with UC-style guarantees.

Security type systems can also help prove security. An information-flow type system can ensure proper combination of specific cryptographic primitives [25, 26]. Owl [27] uses a security type system with built-in primitives to enforce computational guarantees. While these builtins do not fit our goal of abstracting (any) primitive via functionalities, they provide a meaningful level of composition and are worth investigating. In general, type-systems are fast and compositional.

D. Robust Compilation

Abate et al. [2] introduced a large hierarchy of RC criteria that have been used to reason about the security of compilers that preserve memory safety [1, 41], absence of speculation leaks [42], and cryptographic constant time [33, 34]. They require source and target languages to have the same trace model, though they show how to lift that limitation in subsequent work [3]. While we also require our languages to share a trace model, we make that choice for simplicity. The same approach should generalize our result as well. Interestingly, lifting the same limitation in UC would let us formalize the existence of two distinct, but related, environments, one in the real and one in the ideal world. We are unaware of a notion of UC with distinct environments; after all, they are all just (interactive) Turing Machines.

VIII. CONCLUSION

In this work, we have generalized the connection between UC and RC to the computational setting, and then generalized the connection further to arbitrary indistinguishability relations and predicates over programs. We showcased the benefits of this expanded connection by using CRYPTOVERIF, a tool for proving computational indistinguishability, to mechanize a proof of UC security for the Wireguard protocol.

These results let us conclude that CRYPTOVERIF can be used to modularize cryptographic proofs, which are currently very monolithic. We believe our demonstrated connections will extend similar benefits to other tools, though we leave this investigation for future work.

Additionally, there exists a compiler from CRYPTOVERIF into OCaml [14] that the authors proved, already in 2013, robustly preserves hyperproperties. Though the RC framework was only developed years later, the result is equivalent. Since translating from the ideal functionality into the protocol is a compiler, we can create a toolchain that translates CRYPTOVERIF ideal functionalities all the way to executable OCaml protocols. This toolchain would combine properties of sequential composition of compilers [34] and our results to ensure end-to-end guarantees of UC security for real-world executable protocols. However, as we hint at the end of Section VI, a module system in CRYPTOVERIF would streamline the creation of this toolchain, which we also leave for future work.

ACKNOWLEDGMENTS

Thanks to the anonymous reviewers for their insightful comments and suggestions. This work was partially supported by a gift from: the Italian Ministry of Education through funding for the Rita Levi Montalcini grant (call of 2019).

REFERENCES

[1] C. Abate, A. Azevedo de Amorim, R. Blanco, A. N. Evans, G. Fachini, C. Hriţcu, T. Laurent, B. C. Pierce, M. Stronati, and A. Tolmach, "When good components go bad: Formally secure compilation despite dynamic compromise," in *25th ACM Conference on Computer and Communication Security (CCS '18)*, Oct. 2018.

[2] C. Abate, R. Blanco, D. Garg, C. Hriţcu, M. Patrignani, and J. Thibault, "Journey beyond full abstraction: Exploring robust property preservation for secure compilation," in *32nd IEEE Computer Security Foundations Symposium (CSF '19)*, Jun. 2019.

[3] C. Abate, R. Blanco, Ş. Ciobâcă, A. Durier, D. Garg, C. Hriţcu, M. Patrignani, E. Tanter, and J. Thibault, "An extended account of trace-relating compiler correctness and secure compilation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 43, no. 4, Nov. 2021.

[4] C. Abate, P. G. Haselwarter, E. Rivas, A. V. Muylder, T. Winterhalter, C. Hriţcu, K. Maillard, and B. Spitters, "SSProve: A foundational framework for modular cryptographic proofs in Coq," in *34th IEEE Computer Security Foundations Symposium (CSF '21)*, Jun. 2021.

[5] M. Abboud and T. Prest, "Cryptographic divergences: New techniques and new applications," in *Security and Cryptography for Networks (SCN)*, 2020.

[6] C. Acay, R. Recto, J. Gancher, A. C. Myers, and E. Shi, "Viaduct: An extensible, optimizing compiler for secure distributed programs," in *42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '21)*, Jun. 2021.

[7] M. Backes, B. Pfitzmann, and M. Waidner, "The reactive simulatability (RSIM) framework for asynchronous systems," *Information and Computation*, vol. 205, no. 12, pp. 1685–1720, Dec. 2007.

[8] D. Baelde, S. Delaune, C. Jacomme, A. Koutsos, and S. Moreau, "An interactive prover for protocol verification in the computational model," in *42nd IEEE Symposium on Security and Privacy (IEEE S&P '21)*, May 2021.

[9] S. Bai, T. Lepoint, A. Roux-Langlois, A. Sakzad, D. Stehlé, and R. Steinfeld, "Improved security proofs in lattice-based cryptography: Using the Rényi divergence rather than the statistical distance," *Journal of Cryptology*, vol. 31, no. 2, pp. 610–640, Apr. 2018.

[10] G. Barthe, B. Grégoire, S. Heraud, and S. Z. Béguelin, "Computer-aided security proofs for the working cryptographer," in *31st International Cryptology Conference (CRYPTO '11)*, Aug. 2011.

[11] D. Basin, A. Lochbihler, U. Maurer, and S. R. Sefidgar, "Abstract modeling of system communication in constructive cryptography using CryptHOL," in *34th IEEE Computer Security Foundations Symposium (CSF '21)*, Jun. 2021.

[12] B. Blanchet, "An efficient cryptographic protocol verifier based on prolog rules," in *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, Jun. 2001.

[13] ——, "CryptoVerif: Cryptographic protocol verifier in the computational model," 2022, accessed May 2023. [Online]. Available: https://bblanche.gitlabpages.inria.fr/CryptoVerif/

[14] D. Cadé and B. Blanchet, "Proved generation of implementations from computationally secure protocol specifications," *Journal of Computer Security (JCS)*, vol. 23, no. 3, pp. 331–402, 2015.

[15] J. Camenisch, S. Krenn, R. Küsters, and D. Rausch, "iUC: Flexible universal composability made simple," in *25th International Conference on The Theory and Application of Cryptology and Information Security (AsiaCrypt '19)*, Dec. 2019.

[16] R. Canetti, "Universally composable security: a new paradigm for cryptographic protocols," in *42nd IEEE Symposium on Foundations of Computer Science (FOCS '01)*, Oct. 2001.

[17] R. Canetti and M. Fischlin, "Universally composable commitments," in *21st International Cryptology Conference (CRYPTO '01)*, Aug. 2001.

[18] R. Canetti, A. Stoughton, and M. Varia, "EasyUC: Using EasyCrypt to mechanize proofs of universally composable security," in *32nd IEEE Computer Security Foundations Symposium (CSF '19)*, Jun. 2019.

[19] E. Cecchetti, A. C. Myers, and O. Arden, "Nonmalleable information flow control," in *24th ACM Conference on Computer and Communication Security (CCS '17)*, Oct. 2017.

[20] V. Cheval, S. Kremer, and I. Rakotonirina, "DEEPSEC: Deciding equivalence properties in security protocols theory and practice," in *39th IEEE Symposium on Security and Privacy (IEEE S&P '18)*, May 2018.

[21] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security (JCS)*, vol. 18, no. 6, pp. 1157–1210, 2010.

[22] C. Dwork, "Differential privacy," in *33rd International Colloquium on Automata, Languages, and Programming (ICALP '06)*, Jul. 2006.

[23] EasyCrypt Development Team, "EasyCrypt: Computer-aided cryptographic proofs," 2023. [Online]. Available: https://github.com/EasyCrypt/easycrypt

[24] M. Fischlin, F. Günther, B. Schmidt, and B. Warinschi, "Key confirmation in key exchange: A formal treatment and implications for TLS 1.3," in *37th IEEE Symposium on Security and Privacy (S&P '16)*, May 2016.

[25] C. Fournet and T. Rezk, "Cryptographically sound implementations for typed information-flow security," in *35th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '08)*, Jan. 2008.

[26] C. Fournet, J. Planul, and T. Rezk, "Information-flow types for homomorphic encryptions," in *18th ACM Conference on Computer and Communication Security (CCS '11)*, Oct. 2011.

[27] J. Gancher, S. Gibson, P. Singh, S. Dharanikota, and B. Parno, "Owl: Compositional verification of security protocols via an information-flow type system," in *44th IEEE Symposium on Security and Privacy (S&P '23)*, May 2023.

[28] J. A. Goguen and J. Meseguer, "Security policies and security models," in *3rd IEEE Symposium on Security and Privacy (S&P '82)*, Apr. 1982.

[29] O. Goldreich, *Foundations of Cryptography: Volume 1*. New York, NY, USA: Cambridge University Press, 2006.

[30] D. Hofheinz and V. Shoup, "GNUC: A new universal composability framework," *Journal of Cryptology*, vol. 28, no. 3, pp. 423–508, Oct. 2015.

[31] D. Hofheinz, D. Unruh, and J. Müller-Quade, "Polynomial runtime and composability," *Journal of Cryptology*, vol. 26, no. 3, pp. 375–441, Jul. 2013.

[32] Isabelle Development Team, "Isabelle/HOL proof assistant," 2022, accessed May 2023. [Online]. Available: https://isabelle.in.tum.de/

[33] M. Kolosick, B. A. Shivakumar, S. Cauligi, M. Patrignani, M. Vassena, R. Jhala, and D. Stefan, "Robust constant-time cryptography," in *6th Workshop on Principles of Secure Compilation (PriSC '23)*, Jan. 2023. [Online]. Available: https://kolosick.com/robust-crypto-prisc.pdf

[34] M. Kruse, M. Backes, and M. Patrignani, "Secure composition of robust and optimising compilers," Tech. Rep. arXiv:2307.08681, Jun. 2023. [Online]. Available:

https://arxiv.org/abs/2307.08681

[35] R. Küsters, M. Tuengerthal, and D. Rausch, "The IITM model: a simple and expressive model for universal composability," *Journal of Cryptology*, vol. 33, no. 4, pp. 1461–1584, Jul. 2020.

[36] K. Liao, M. A. Hammer, and A. Miller, "ILC: A calculus for composable, computational cryptography," in *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*, Jun. 2019.

[37] B. Lipp, B. Blanchet, and K. Bhargavan, "A mechanised cryptographic proof of the WireGuard virtual private network protocol," in *4th IEEE European Symposium on Security and Privacy (EuroS&P '19)*, Jun. 2019.

[38] J. McLean, "Proving noninterference and functional correctness using traces," *Journal of Computer Security (JCS)*, vol. 1, no. 1, pp. 37–57, Jan. 1992.

[39] D. Micciancio and M. Walter, "Gaussian sampling over the integers: Efficient, generic, constant-time," in *27th International Cryptology Conference (CRYPTO '07)*, Aug. 2017.

[40] M. Patrignani, "Why should anyone use colours? or, syntax highlighting beyond code snippets," *CoRR*, vol. abs/2001.11334, 2020. [Online]. Available: https://arxiv.org/abs/2001.11334

[41] M. Patrignani and D. Garg, "Robustly safe compilation, an efficient form of secure compilation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 43, no. 1, Feb. 2021.

[42] M. Patrignani and M. Guarnieri, "Exorcising spectres with secure compilers," in *28th ACM Conference on Computer and Communication Security (CCS '21)*, Nov. 2021.

[43] M. Patrignani, R. Künnemann, and R. S. Wahby, "Universal composability is robust compilation," Tech. Rep. arXiv:1910.08634, Dec. 2022. [Online]. Available: https://arxiv.org/abs/1910.08634

[44] A. Petcher and G. Morrisett, "The foundational cryptography framework," in *4th Principles of Security and Trust (POST '15)*, Apr. 2015.

[45] A. Rastogi, M. A. Hammer, and M. Hicks, "Wysteria: A programming language for generic, mixed-mode multiparty computations," in *35th IEEE Symposium on Security and Privacy (S&P '14)*, May 2014.

[46] A. Rastogi, N. Swamy, and M. Hicks, "Wys*: A DSL for verified secure multi-party computations," in *8th Principles of Security and Trust (POST '19)*, Apr. 2019.

[47] A. Roscoe, "CSP and determinism in security modelling," in *16th IEEE Symposium on Security and Privacy (S&P '95)*, May 1995.

[48] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 1, pp. 5–19, Jan. 2003.

[49] ——, "A model for delimited information release," in *International Symposium on Software Security*, Nov. 2003.

[50] I. Sweet, D. Darais, D. Heath, R. Estes, W. Harris, and M. Hicks, "Symphony: Expressive secure multiparty computation with coordination," in *7th International Conference on the Art, Science, and Engineering of Programming (⟨Programming⟩ '23)*, Mar. 2023.

[51] Y. Wang, Z. Ding, G. Wang, D. Kifer, and D. Zhang, "Proving differential privacy with shadow execution," in *40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*, Jun. 2019.

[52] Y. Wang, Z. Ding, Y. Xiao, D. Kifer, and D. Zhang, "DPGen: Automated program synthesis for differential privacy," in *28th ACM Conference on Computer and Communication Security (CCS '21)*, Nov. 2021.

[53] D. Wikström, "Simplified universal composability framework," in *13th IACR Theory of Cryptography Conference (TCC '16)*, Jan. 2016.

[54] S. Zdancewic and A. C. Myers, "Robust declassification," in *14th IEEE Computer Security Foundations Workshop (CSFW '01)*, Jun. 2001.

[55] ——, "Observational determinism for concurrent program security," in *16th IEEE Computer Security Foundations Workshop (CSFW '03)*, Jun. 2003.

[56] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers, "Secure program partitioning," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 3, pp. 283–328, Aug. 2002.

[57] D. Zhang and D. Kifer, "LightDP: Towards automating differential privacy proofs," in *44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL '17)*, Jan. 2017.

[58] L. Zheng, S. Chong, A. C. Myers, and S. Zdancewic, "Using replication and partitioning to build secure distributed systems," in *24th IEEE Symposium on Security and Privacy (S&P '04)*, May 2003.