From Secure Compilers to Robust Abstractions:

A general theory of secure compilation and reduction proofs

Emiel Lanckriet
KU Leuven
Leuven, Belgium
emiel.lanckriet@kuleuven.be

Dominique Devriese
KU Leuven
Leuven, Belgium
dominique.devriese@kuleuven.be

Frank Piessens
KU Leuven
Leuven, Belgium
frank.piessens@kuleuven.be

ABSTRACT

An open system P robustly satisfies a property H if H holds for any context that P is linked with. This concept of robust satisfaction of properties and the preservation of such properties by relations appears in a lot of places in computer science. They most prominently feature in the field of secure compilation, but they are also clearly present in fields such as cryptography and computability theory. Starting from this observation, we build a unifying framework that allows to more easily see the links between these fields and think at a higher level of abstraction about each of these. Concretely, we introduce a general structure of system models which contains the essential aspects to talk about robust properties, namely programs/problems and contexts/solvers. In this structure of system models, we then define what it means for a robust property of one system to be preserved or ensured for a related system. Finally, we show how several well known concepts from the literature are simple instantiations of concepts in our framework.

KEYWORDS

secure compilation, reduction proofs, computability theory, cryptography

1 INTRODUCTION

Roughly speaking, a property of a system is robust [12] [6] if it holds for all possible contexts that the system can interact with. Robust properties are of importance in many guises and in many areas of computer science, often in security (where the context that one quantifies over can be thought of as the attacker) but also in other disciplines. For instance, in programming language theory, contextual equivalence is robust equi-termination [1]. In computability theory, undecidability of a language is robust non-decidability (a specific decider cannot decide the language) [15]. In cryptography, game-based security definitions say that it is robustly hard to win the corresponding game [16]. (We will elaborate on all these examples later in the paper.)

Similarly, relating the robust properties of one system with those of another related system is studied in many forms. Fully abstract compilation is the preservation and reflection of contextual equivalence [1]. Secure compilation can be formalised as the preservation of robust properties (including, for instance, non-interference) [6]. In computability theory, reductions relate the undecidability of one language to that of another language [15] and protocol security in cryptography is proven by relating security games [16].

Despite the high-level similarities, all these different instances of robust properties and the relations between them also differ in subtle aspects, and hence are not widely recognised as formally similar notions. The systems under consideration can be deterministic (e.g. for non-interference) or probabilistic (e.g. for cryptographic security), and can be thought of as programs (e.g. in secure compilation) or as languages (e.g. in computability theory). The robust property being considered can be a safety property or a liveness property, a trace property or a hyperproperty. The contexts that are quantified over can interact with single systems (e.g. non-interference), with pairs of systems (e.g. contextual equivalence), or even with distributions of systems (e.g. cryptography). The relation between robust properties of systems can be the preservation of the same property, or they can relate different properties. Preservation of robust properties can be required only for specific programs or a single security game, but may also apply to arbitrary programs translated by a 'secure compiler' or more generally, related by a 'secure abstraction' relation.

The objective of our work is to propose a novel unifying framework that makes the connections between all the examples above precise. We propose definitions for a general notion of robust property (formulated within an abstract structure of "system models") in which all the examples above (and many more) fit naturally. We then study relations between these system models with a generalised notion of robust property preservation. One can think of the framework as a general and abstract study of the notions of secure compilation and reduction proofs (or proofs by back-translation).

We believe our framework is useful, both for providing deeper insights in the field of secure compilation and its connections with other research fields, as well as for reusing theorems and proofs across a variety of disciplines. In this work-in-progress paper, we define the framework, instantiate it to a wide range of examples, and give preliminary examples of general useful theorems at the abstraction level of the framework. In Section 2, we introduce several motivating examples that we used to guide the design of this framework. In Section 3, we define our notion of system model, what it means for a property to hold robustly and we show how our guiding examples can be written in our framework. Section 4 introduces the notion of (F, H_1, H_2)-preserving which is our general notion of robust property preservation. Subsequently, in Section 5, we demonstrate for each of the guiding examples how to formulate them as system models and how such 'secure abstractions' show up (often they are relations). We explain how our framework generalises and improves existing proposals (particularly in the field of secure compilation) in Section 6.

2 MOTIVATING EXAMPLES IN MORE DETAIL

As discussed above, we used several examples to motivate the design of our framework. In this section, we will introduce these examples in more detail, such that, later on, it is clear how they fit into the framework and why they are interesting guiding examples.

2.1 Computability theory

Our first example of a system with robust properties is computability theory [15]. In computability theory, we consider a set of strings A called a language and we say that a certain Turing machine M decides A if M always halts and outputs fail if the input is not an element of A and it succeeds if the input is an element of A. We call a language decidable if there exists a Turing machine that decides it and similarly we call a language undecidable if no Turing machine decides it. Notice that if we consider the language as being an open system [1] and the Turing machine as a context, then being undecidable is a robust property because it (the machine does not decide the language) holds under any choice of Turing machine. In this case, we can then consider the deciders as the contexts and a property of an open system/language is robust if it holds for all possible contexts/deciders.

Suppose we know that A is undecidable and we want to prove that B is undecidable. Then, the relation relating A to B preserves a robust property, namely undecidability. To prove the preservation of this property, we can use contradiction and show that if B has a decider, then we can construct a decider for A. This proof shows that if A is undecidable, then B is undecidable, because if B were decidable, then we would also be able to decide A. This proof technique is called proof by reduction and it appears frequently in proofs of preservation of robust properties.

2.2 Complexity theory

Complexity theory [15] is very similar to computability theory, but there is a restriction on the Turing machines that can function as deciders if you want to show that a language belongs to the complexity class P. In other words, less contexts are considered, hence it is easier for a property to hold robustly. Specifically, a Turing machine has a time complexity $f : \mathbb{N} \to \mathbb{N}$ if for any input of size n the Turing machine takes at most $f(n)$ steps. We use TM-Time(f) to denote the class of Turing machines with a time complexity of $O(f)$, which means that asymptotically the time complexity grows slower then f. The set of Turing machines that finishes in time polynomial in the size of the input is denoted by TM-P = $\bigcup_{k\in\mathbb{N}}$ TM-Time($n \mapsto n^k$). Now, the class P is the set of problems that are decided by a Turing machine in TM-P. Note that we can make a class of languages in the same way for any complexity class of Turing machines. In the same way as for computability theory, we can prove that $A \notin P$ implies $B \notin P$ by assuming that there exist a polynomial time decider for B and constructing a polynomial decider for A from it.

Also, note that we can prove $P \subset DEC$, where DEC is the set of all decidable languages, by reduction. Concretely, we find that A is undecidable implies $A \notin P$, because if A were in P then the polynomial-time decider of A would also be a decider of A, hence A would be decidable.

2.3 Cryptographic games

In Section 2.1, we introduced a system with robust properties and a relation that preserves a robust property. In this section, we present a systems that have robust properties and relation between these systems. Contrary to the previous sections, these system also contains probabilistic problems. In this case, we call the problems cryptographic games. The information in this section is taken from [16].

Cryptographic games are specific challenges built by cryptographers to show that a certain cryptographic protocol provides the claimed security by showing that no feasible attacker can solve the challenge. Specifically, a game is a program (the language is mostly pseudocode, but there are tools such as EasyCrypt [7] that use a more formal language) that can make probabilistic choices. This program then links with an adversary and provides it with oracles. Oracles are (sometimes stateful) programs that the adversary can call at any point in its execution. Most of the time, execution is handed over to the adversary after setting up the protocol and at the end the adversary sends an answer back to the game, the game checks the answer and outputs whether or not the adversary win the game. Often, a game depends on the probabilistic choice of some key, and the length of this key (the security parameter) determines how hard it is to win the game. This is why most games are parametric in a security parameter. As such, game-based security definitions are robust properties of $\mathbb{N}$-indexed families of probabilistic programs. In this case, we say that the arity[2] of cryptographic games is sequences of distributions.

An example of such a cryptographic game is one-way security for symmetric encryption schemes (depicted in Figure 1). First, imagine that the oracles O_{e_k} and O_{d_k} are not present, in which case we get the OW-PASS security game (one-way security against passive attackers). This game chooses a key and a message randomly and sends the message encrypted with the key to the adversary. The adversary wins if it can send back the original message, because this means the encryption does not protect the message. However, a real adversary could have access to more resources, for example, the adversary might be able to trick the sender to encrypt arbitrary attacker-chosen messages and observe the ciphertext. To express that such extra power should not break the encryption we can design the game such that it hands a decryption and encryption oracle (a program that decrypts/encrypts any value passed to it) to the adversary and require that the adversary can still not win the game. This game is called OW-CCA (chosen ciphertext attack) (cf. Figure 1). Even stronger security can be defined by adding other relevant oracles.

To prove that the adversary cannot win the game, the game is generally reduced to the unwinnability of another game. Concretely, well-known hard problems are commonly used for this, such as factoring primes or performing discrete logarithms. Reduction of game G_1 to game G_2 here means that if there exists some feasible adversary that can win game G_2 with a significant probability, then we can construct a feasible adversary that can win G_1 with a significant probability. Similarly to computability theory, if this can be proven, then the unwinnability of G_1 by a feasible attacker implies the unwinnability of G_2.

In the previous paragraph, the words feasible and significant were used a few times. Briefly, each game and its adversary have

[1] One could also consider the string as the open system and consider undecidability as a set-ary robust property (robust property on sets of strings), but the secure abstraction doesn't relate individual strings in A to individual strings in B. However, mapping reducibility [15] is similar to this approach.

[2] In the context of this work, the arity of property is not limited only to the number of elements it talks about, but it might also denote that the property is about a distribution or a function of such elements.

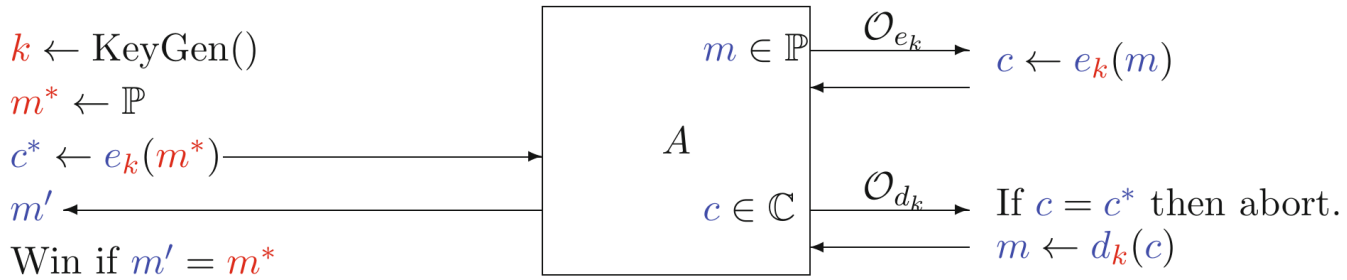

Figure 1: Security game for symmetric key OW-CCA. Taken from [16].

an input which is the security parameter (this determines e.g. the key size) and an adversary is feasible if it can run in a time polynomial in the security parameter. A probability (that is dependent on the security parameter) on the other hand is insignificant if it decreases faster then any inverse polynomial of the security parameter. Interesting to note is that the game should also be feasible. The above-mentioned hard problem will then also depend on the same security parameter.

The idea of a cryptographic game is the same for all games. However, there is a prevalent type of game that is played because it is generally easier to prove things about. These are indistinguishability games where the game makes a one-bit choice randomly and based on this gives one oracle or another to the adversary and the adversary has to guess which oracle it received. For example, one choice is a proper encryption scheme and the other just returns random values. If the adversary cannot win this game with a significant probability, then this is considered evidence that the adversary (with the same oracles) cannot break the encryption scheme: they can not even distinguish proper ciphertexts from random values. It is interesting to note here that these indistinguishability games feel very similar to contextual equivalence in secure compilation research.

2.4 Secure compilation criteria

The previous sections introduced preservation of robust properties for a relation between single open systems. However, in programming language research, e.g., in the field of secure compilation, we often look at whole programs that output traces. To quote [4], "a trace is an ordered sequence of events—such as inputs from and outputs to an external environment—that are produced by the execution of a program.". In this instance, they are talking specifically about traces in CompCert and CakeML, two verified compilers, but the core idea of traces is the same in most other cases as well. The main idea is that a whole program, that is a partial program linked with a context, can have a more complicated and interesting interaction with the environment. These trace-producing programs and contexts can be written in any language with a trace-producing semantics.

The trace model is the most common model in the field of secure compilation where several interesting examples for our framework reside. First of all, secure compilation studies compilers that preserve robust properties, hence the relation that preserves robust properties is a function in this case. Contrary to previous sections, this compiler can map any program, hence the question of preservation is not about one problem as it was in Sections 2.1, 2.2 and 2.3, but about all programs in the source language.

The first and most prominent secure compilation criterion is full abstraction, this is the preservation and reflection of contextual equivalence. Two program P and Q are contextually equivalent if the result of coupling them both with the same context C is equivalent for any choice of context. In symbols, contextual equivalence is $P \equiv_{ctx} Q \iff \forall C : C[P] \equiv C[Q]$ where $C[P]$ means linking context C with program P. Equivalence, in this case, can refer to any relevant equivalence relation, such as equi-termination or having the exact same traces. Full abstraction differs from the previous examples, because it describes the preservation of a property about pairs of programs instead of single programs. In this case, we say that the arity is pairs.

Another prominent secure compilation criterion is RHP [6] which denotes that a compiler preserves all robust hyperproperties. A hyperproperty is a property about the set of all possible traces a (possibly nondeterministic) program can have. Hyperproperties allow one to describe criteria, such as non-interference which are not properties (of a single execution trace), because they depend on multiple (two in this case) different executions of the same program. A program P satisfies a robust hyperproperty if P linked with any context C satisfies the hyperproperty. In symbols, we write this as the compiler $[\![\cdot]\!]$ satisfies RHP if and only if $\forall H : \forall P : (\forall C_S : \mathit{behav}(C_S[P]) \in H) \implies (\forall C_T : \mathit{behav}(C_T[[\![P]\!]]) \in H)$ where H can be any hyperproperty, $\mathit{behav}()$ maps whole programs to their sets of possible traces ,and C_S and C_T range over respectively all source contexts and all target contexts. Note that *RHP* does not consider the preservation of one (hyper-)property as we did before, but the preservation of a whole class of (hyper-)properties. However, because the property of the source and target language are exactly the same, we need to make sure that the trace semantics of our source and target language are exactly the same. Follow-up work by Abate et al. [5] did relax this constraint.

Full abstraction preserves a robust binary hyperproperty while RHP preserves all unary hyperproperties. Abate et al. [6] also introduce the preservation of properties for other arities as well, such as RrHP which specifies that the compiler should preserve all relational hyperproperties. Relational hyperproperties are properties

about sequences of sets of traces. Concretely, robust relational hyperproperty then means that for every context C they consider the sets of execution traces of sequences of partial programs each linked with C. Similarly, one can also discuss preservation of properties about k sets of traces and this is called RKrHP.

Just as proofs by reduction are very common to prove robust properties in computability theory or about cryptographic games, proofs by back-translation are very common in secure compilation. Essentially, one tries to back-translate the context C_T that interacts with the compiled program $[\![P_S]\!]$ in the target language back to the source. This back-translation, $bcktr(C_T)$, should be constructed in such a way that interaction with the original program, P_S behaves similar to $C_T[[\![P_S]\!]]$. So, $bcktr(C_T)[P]$ should behave similarly to $C_T[[\![P_S]\!]]$. The meaning of similarly depends on the kind of criterion that you are trying to prove. Note, how proofs by back-translation are very similar to proofs by reduction where we basically back-translate a decider of B to a Turing machine that decides A.

2.5 Computational soundness of cryptography

The previous section discussed preservation of robust properties by compilers, but probabilistic language were left out of scope and the properties always had the same arity (e.g. pairs to pairs, sequences to sequences). In this section, we introduce a compiler from a deterministic symbolic to a probabilistic computational language. Note that this means that source and target have different arities, from single terms to distributions of terms. Concretely, we discuss the work by Abadi and Rogaway [3] about the computational soundness of symmetric encryption. For this compiler, they show some kind of full abstraction.

The symbolic language is an expression language with encryption with a key, denoted as $\{M\}_K$. The indistinguishability relation between expressions is defined as having the same pattern where a pattern replaces each encrypted term by $\square$ if the key cannot be derived from the expression. For example, $(K, \{M\}_K)$ remains $(K, \{M\}_K)$ because $\{M\}_K$ can be decrypted as K can be learned, but $(0, \{M\}_K)$ becomes $(0, \square)$. Hence, $\{M\}_{K_1}$ is indistinguishable from $\{N\}_{K_2}$ because for both the pattern is $\square$.

The computational language of Abadi and Rogaway is once again an expression language, but instead of keys and an encrypt expression, a triple of functions called $\Pi = (\mathcal{K}, \mathcal{E}, \mathcal{D})$ is given. The functions in Π are the following:

$$\mathcal{K} : \text{Parameter} \times \text{Coins} \to \text{Key}$$
$$\mathcal{E} : \text{Key} \times \text{String} \to \text{Ciphertext}$$
$$\mathcal{D} : \text{Key} \times \text{String} \to \text{Plaintext}$$

where $\mathcal{K}$ is the key-generation algorithm, $\mathcal{E}$ is the encryption algorithm and $\mathcal{D}$ is the decryption algorithm. The Coins argument of $\mathcal{K}$ is the source of randomness (a random sequence of 0's and 1's as long the security parameter) for the key generation and omitting them as an argument of $\mathcal{K}$ indicates that $\mathcal{K}$ outputs the distribution of keys that results supplying a uniform distribution of coins. Lastly, encrypting and decrypting m with the same key should return m again.

Attackers in this language are probabilistic polynomial-time Turing machines meaning they have access to probabilistic choices and they run in a time that is polynomial in the security parameter that determines the key length. Attackers receive one value and output either 0 or 1 based on this value. In this language, we say that an attacker A cannot distinguish two expressions M and N (which can be distributions) if

$$\epsilon(\eta) = |\mathbb{P}(x \leftarrow M : A(\eta, x) = 1) - \mathbb{P}(x \leftarrow N : A(\eta, x) = 1)|$$

is a negligible function of η which means that for all $c \in \mathbb{N}$, it holds that ϵ is $O(\eta \to \eta^{-c})$. Now, M and N are considered indistinguishable if no probabilistic polynomial-time adversary A can distinguish M and N. The notation $x \leftarrow M$ means we sample the distribution M and x is the result.

Abadi and Rogaway [3] construct a translation $[\![\cdot]\!]_\Pi$ from the symbolic model to the computational model and they showed that if M is indistinguishable from N (they have the same pattern), then $[\![M]\!]_\Pi$ is indistinguishable from $[\![N]\!]_\Pi$ under the condition that Π is a type-0 secure encryption scheme. Type-0 security means that

$$|\mathbb{P}(k, k' \leftarrow \mathcal{K}(\eta) : A^{\mathcal{E}_k(\cdot), \mathcal{E}_{k'}(\cdot)} = 1) - \mathbb{P}(k \leftarrow \mathcal{K}(\eta) : A^{\mathcal{E}_k(0), \mathcal{E}_k(0)}) = 1|$$

is a negligible function of η for every probabilistic polynomial-time adversary A.

2.6 Probabilistic secure compiler for layout randomisation

In the previous section, we presented a compiler that changes the arity (from unary to security parameter-indexed families of distributions). This language section we present a compiler that also changes the arity, but the considered programming language is more interesting. Whole programs can be executed, resulting in either fail (terminate with an error), diverge or evaluate to a value. Furthermore, the languages are typed, which means that not all contexts can be linked with all programs.

Specifically, in [2], Abadi and Plotkin show the security of layout randomisation by creating two languages. Both languages are typed lambda calculi extended with locations to store values. The source language of the compiler is the high-level language where public locations are free to use for anyone and private locations are inaccessible to the attacker (if the program does not hand the pointers to private locations explicitly). The target language is the low-level language where all locations (even the private ones) can be accessed by guessing the right memory location (a natural number). However, some memory locations are not mapped and if a program tries to access an unmapped location, the program throws a fatal error. The intuition here is that layout randomisation protects the private memory, because an attacker who guesses addresses to try to access private memory is very likely to first access unmapped memory before finding the private memory (because the layout is randomised and the address space is sufficiently large). Of course, since in the low-level model we talk about the probability of hitting the private memory, we need to add some randomness to our programming language. This randomness is added by parametrising a program over its memory layout which maps names of locations to integers in the memory range. Such a memory layout is then chosen randomly.

The security of the layout randomisation is now shown by proving that two high-level programs that are indistinguishable to any

context can be compiled to two programs that are indistinguishable to any context. Indistinguishable in the target language means that linked with the context the closed system either evaluates to the same value, they both diverge or they both have a sufficiently high probability to throw an error. A high probability means for a large fraction of the possible choices of memory layout (these are normally uniformly distributed). Specifically, a high probability means a probability that is higher or equal to the probability that a randomly chosen non-public memory location is not private memory (so unmapped), so if private memory is a sufficiently small fraction of the total memory, then this probability is high. In essence, this says that either two programs have the same output or there is a very high probability for both programs that the attacker's memory access will result in a fatal error, hence the probability that the attacker can distinguish the programs is negligible. Notice that this indistinguishability is again a robust property and the preservation result is actually full abstraction, but the notions of contextual equivalence in source and target are very different as the contextual equivalence in the target is defined probabilistically.

An interesting thing to note in this work is that their proof uses some kind of back-translation and the back-translation can readily give a stronger criterion than full abstraction, namely something akin to a probabilistic version of RrHP [6], so the arity of the criterion is sequences. However, the output of these programs are not traces, but just one value, error or it diverges.

2.7 Universal composability

In the Section 2.3, we presented game-based cryptographic proofs which state a challenge and if this challenge can not be won, then the protocol is deemed secure. However, it turns out that these security guarantees do not necessarily remain true when you start composing it with other cryptographic primitives or multiple instance of the primitive. For example, reusing the key or some nonce might break your security guarantees and to detect this with a game, you need to test that exact scenario.

To combat this, the framework of universal composability (UC) was devised which was originally introduced in [8], but we will follow the restricted model presented in [9]. This framework does have a fixed formal language, namely the language of interactive Turing machines (ITMs). ITMs are systems of Turing machines that all have a dedicated input tape and the Turing machines can write on each others tape. By writing to another tape, execution is given to the recipient. In this way, execution remains deterministic which makes it easier to reason about. Besides these input tapes, each Turing machine also has a dedicated backdoor tape and a subroutine output tape. A backdoor tape is used to model a potential backdoor that your protocol may have and only the dedicated adversary is allowed to write on this tape. The backdoor tape is for example used to send messages which corrupt the party (a set of ITMs representing a party in the protocol). The subroutine output tape is used to receive output from ITMs that are considered a subroutine.

Now, we can model a protocol such as encryption by a set of ITMs that represent the two (or more) parties of the communication and sent possible backdoor information to the dedicated adversary and are corruptable by the adversary. This dedicated adversary is itself an ITM that can only write to backdoor tapes, receives all the backdoor messages and can send messages to a specific ITM called the environment.

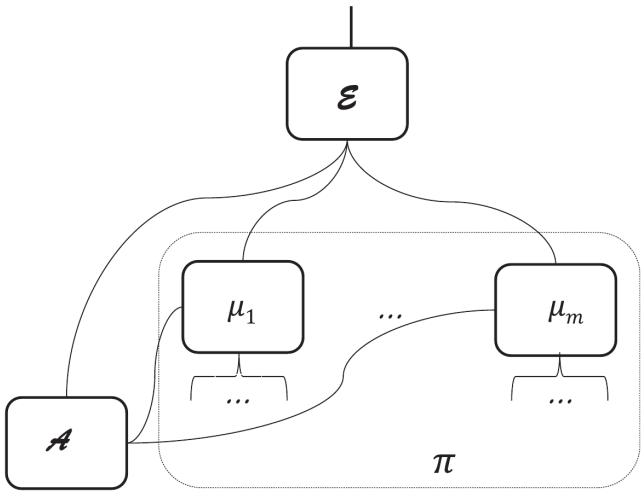

Figure 2: Picture from [9] displaying a protocol π (made up of $\mu_1, \ldots, \mu_m$ and subroutines), an adversary A and an environment E in communication with each other.

Now, to prove security of a certain cryptographic protocol, a cryptographer can model the protocol using ITMs and send the right backdoor information to the adversary. Subsequently, he can devise an ideal functionality which is a protocol that models how one expects a secure run of the protocol to go (e.g. send a random message over the public channel and the actual message over a private channel to model perfect encryption). By then showing that for every protocol coupled with an adversary, we can find a simulator that takes the role of the adversary with regards to the ideal functionality. This simulator should be chosen such that some third set of ITMs, the environment machine, which starts execution cannot distinguish between these two couples of a protocol and an adversary. In essence, this means that if the adversary can leak certain backdoor information, then we can create indistinguishable backdoor information just from the intended leakage of the ideal functionality. If a protocol ϕ is such that for every adversary of π there is a simulator, such that no environment can distinguish ϕ and the simulator from π and the adversary, then we say that π UC-emulates ϕ. Figure 2 shows a triple of a protocol, adversary and environment that communicate with one another. Important on this picture is that coming in from the side means backdoor input/output, while top/bottom is normal input/output.

In this system of ITMs, we can ask the question what an efficient adversary and environment are. These should be ITMs that run in polynomial time, but polynomial in what variable. To solve this, [9] devised a compositional type of polynomial runtime where each communication step also gives along some import and the total runtime of an ITM should be smaller then some polynomial in the difference between the import received and sent. Then, the whole system will run in time polynomial in the import originally given to the environment machine. To make this whole system depend on a security parameter k, we start by instantiating the protocols in a

principled way for each choice of k. The initial import given to the environment should be some polynomial of the security parameter. Finally, because the task performed by each ITM can depend on the security parameter, each ITM needs at least k import to start running.

Note how the existence of the simulator corresponds very well to the back-translation of contexts in the secure compilation research or reductions in proofs by reduction. This is a connection that has already been explored and there is work relating the notion of UC-emulation with RHP [13].

Lastly, the fact π UC-emulates ϕ is the basic notion in a UC framework, however, it does not really say anything about composition. That is what the universal composition theorem states. We state it for the restricted model of [9] because there are some details involving identities from the general model that we do not discuss. The universal composition theorem states that if ϕ is a subroutine of some protocol ρ, π UC-emulates ϕ and π is identity-compatible with ρ and ϕ, then protocol $\rho^{\phi \to \pi}$ UC-emulates ρ. Here, the operation $.^{\phi \to \pi}$ denotes replacing all instances of the subroutine ϕ by π in the protocol ρ. This replacement is possible because π is identity-compatible with ρ and ϕ which means that π should have the right identities and communicate with the right identities to emulate ϕ in that regard. The interesting thing is that UC-emulation originally only relates to protocols, but in the universal composition theorem it shows the security of $.^{\phi \to \pi}$ regarded as a secure compiler.

3 SYSTEM MODELS

As demonstrated in the previous section the ideas of robust properties and their preservation show up in several places throughout computer science. To study robust properties, an open system/program/problem and some form of context/attacker/solver are required. Furthermore, we want to make a clear distinction between the structure of open systems and contexts on the one hand and robust properties and their preservation on the other hand. In this section, we define a structure that defines the structure of open systems and contexts and call it a system model. This general definition of open systems and contexts makes it easier to talk about robust properties and their preservation.

A system model defines a set of open systems/programs/problems that can be linked with a form of contexts/attackers/solvers to obtain a full system/whole program/problem solution attempt. A system model also defines how linking a system/program/problem P with a context/attacker/solver C works. This linking function is often denoted by $C \bowtie P$.

Definition 1. A system model, *SysModel*, consists of three parts:

(1) A set of open systems, $\mathcal{P}$,
(2) A set of valid contexts, $\mathcal{C}$,
(3) A linking function, $\bowtie : \mathcal{P} \times \mathcal{C} \to \mathit{ClosedSys}$, which sends a pair of an open system $P \in \mathcal{P}$ and a valid context to some domain of closed systems, *ClosedSys*.

We denote such a system model as $\mathit{SysModel}(\mathcal{P}, \mathcal{C}, \bowtie)$.

In programming languages with a type system (or other types of static specification), linking is often not allowed with arbitrary contexts but only with contexts that have the right type (e.g. [2]). There are also other situations where the set of valid contexts is restricted, for example, certain identities cannot be used by the context or the context cannot use certain variable names to avoid a naming clash. We could accommodate this by introducing a general notion of interfaces, where programs and contexts can be associated to a certain interface and linking only happens if this is the case. To avoid complicating the definition of system models with a notion of interface specification, we consider such settings as a family of system models, indexed by the interface specification. For example, in many programming languages a type system is used and this results in $\mathit{SysModel}^\sigma = \mathit{SysModel}(\mathcal{P}^\sigma, \mathcal{C}^\sigma, \bowtie^\sigma)$ where $\mathcal{P}^\sigma$ is the set of programs with type σ and $\mathcal{C}^\sigma$ is the set of contexts that can link with programs of type σ.

3.1 Robust properties

Within this structure of a system model, we would like to discuss the idea of robust properties. In general, a property H holds robustly for an open system P if H holds for every closed system that results from linking P with an arbitrary context C. In symbols, this is

$$P \in \mathit{robustly}(H) \iff \forall C \in \mathcal{C} : C \bowtie P \in H.$$

However, some properties H that are encountered in the literature do not consider individual open systems, but several open systems at once, e.g. a pair of programs in full abstraction or a set of programs indexed by a security parameter in cryptography; we call this the arity of the property. To model arities that are not inherent to the system model we consider, we can use a functor on sets which can then be applied to the set of programs and the linking of the context. Briefly, a functor is a map on sets that also behaves well on functions, for example in this case the functor can be applied both to the set of open systems and to the function that links with a context. Note that although functor is a category theoretic concept, we do not aim to make a category theoretic framework, such as [17], but simply borrow the concept as it has the behaviour we need in this situation. With this functor, we define

$$P \in \mathit{robustly}(F, H) \iff \forall C \in \mathcal{C} : F(C \bowtie) P \in H$$

where $P \in F(\mathcal{P})$ and $H \in 2^{F(\mathit{ClosedSys})}$ and $(C \bowtie)$ denotes the partially applied linking operator that maps programs to programs linked with context C. Some examples are the functor for pairs which is $X \to X \times X$ and the distribution functor which sends a set of open systems to the set of distributions on that set. In both cases, applying the functor to a function just applies the function pointwise.

4 (F, H_1, H_2)-PRESERVING

Now that we have a notion of the models under consideration and what constitutes a robust property, we can present criteria on these relations that express the preservation of robust properties. Concretely, we will discuss the preservation of robust properties under certain relations. Our proposal has the general form of (F, H_1, H_2)-preserving where F is a functor from Set to Set that denotes the "arity" of the properties H_1 and H_2 about respectively source and target closed systems. Note that, because we are studying relations, not functions, a functor is generally not enough for well-behaved behaviour. In general, it also needs to behave well for mapping related elements. Concretely, we need for all $\mathcal{T} : A \to B \to \mathit{Prop}$

that $F(\mathcal{T}) : FA \to FB \to Prop$ holds and this is the same as $F(R) \subset F(A_1) \times F(A_2)$ if $R \subset A_1 \times A_2$ for $F : Set \to Set$. For example, for pairs this could be defined as $((a,b),(c,d)) \in F(R)$ if $(a,c) \in R$ and $(b,d) \in R$ and for distributions this could specify $(U,V) \in F(R)$ if $\forall P' : \sum_{\{P \mid P\, R\, P'\}} \mathbb{P}_U(P) = \mathbb{P}_V(P')$ or it could mean that distributions U and V have probability 1 for P_1 and P_2 and $P_1\, R\, P_2$ holds. Of course, relations should be mapped consistently with applying the functor to functions. Concretely, if the relation $\mathcal{T}$ is a function, the operation on the relations should behave exactly like the operations on morphisms. Consequently, if our relation is a function, then the theory also goes through for functors as arities (without the extra condition on relations). Besides the demand that operations on sets, functions and relations are consistent, there is freedom to define these in a way that seems fitting for the appropriate security criterion. These conditions on the functor seem to be similar to pro-arrow equipments [14], but we have not looked into this yet.

Below, we give the definition of (F, H_1, H_2)-preserving:

Definition 2. For $SysModel_1$ and $SysModel_2$, a relation $\mathcal{T} \subset \mathcal{P}_1 \times \mathcal{P}_2$ with $H_1 \in 2^{F(ClosedSys_1)}$ and $H_2 \in 2^{F(ClosedSys_2)}$ is (F, H_1, H_2)-preserving if

$$\forall (P_1, P_2) \in F(\mathcal{T}) : P_1 \in robustly(F, H_1) \implies P_2 \in robustly(F, H_2). \tag{1}$$

Concretely, this means that if a program in $SysModel_1$ robustly satisfies a certain property H_1 of arity F, then the related program in $SysModel_2$ will robustly satisfy property H_2 also of arity F. Often, we will denote the relation $\mathcal{T}$ between $\mathcal{P}_1$ in $SysModel_1$ and $\mathcal{P}_2$ in $SysModel_2$ with $\mathcal{T} : SysModel_1 \to SysModel_2$.

The criterion (F, H_1, H_2)-preserving has an equivalent formulation inspired by the proof ideas in proofs by back-translation or proofs by reduction. This criterion is the following:

$$\begin{aligned} \forall (P_1, P_2) \in F(\mathcal{T}) : \forall C_2 \in \mathcal{C}_2 : \exists C_1 \in \mathcal{C}_1 : \\ F(C_2 \bowtie_2)(P_2) \notin H_2 \implies F(C_1 \bowtie_1)(P_1) \notin H_1. \end{aligned} \tag{2}$$

This means that if some target context can make the open system in the target fail the robust property, then there is certainly a source context that makes the original open system fail its robust property as well. In the case of security, if the source program is secure, then this means that the compiled program is secure, because if it were insecure, then the original program would also be insecure. These criteria are inspired by [6].

Note that we could just as well use the contraposition of Equation 2 with $F(C_i \bowtie_i)(P_i) \in H_i$, but this formulation better reflects the reasoning used in reduction proofs.

As mentioned in Section 2.4, there are secure compilation criteria such RHP and RrHP that state that the compiler preserves a whole set of properties. To support this, we define notation for preservation of a relation on properties.

Definition 3. Let $\mathcal{T} : SysModel_1 \to SysModel_2$ be a relation and $R \subseteq 2^{F(ClosedSys_1)} \times 2^{F(ClosedSys_2)}$, then we say that $\mathcal{T}$ is (F, R)-preserving if and only if:

$$\forall (H_1, H_2) \in R : \mathcal{T} \text{ is } (F, H_1, H_2)\text{-preserving}.$$

One idea that we still need to look into is whether it makes sense to apply the functor directly to the relation and the system models instead of only in the preservation criterion. So, instead of stating $\mathcal{T} : SysModel_1 \to SysModel_2$ is (F, H_1, H_2)-preserving, one would state $F(\mathcal{T}) : F(SysModel_1) \to F(SysModel_2)$ is (H_1, H_2)-preserving. This would maybe be a more principled approach, since in the current framework there seems to be a somewhat arbitrary difference between the arity that is added in the security property and the arity that is inherent to the relation. In this new approach, the difference would just be whether the arity can be factored out of the compiler as a functor or not. However, to do this we still need to make some decisions, most importantly, what it means to apply a functor to a system model.

4.1 Implications

Given this structure a natural question to ask is what this gets us. One result we found is that secure compilation is preserved under arity changes that can be modelled by a natural transformation:

Theorem 1. *Suppose the relation $\mathcal{T}$ is (F, H_1, H_2)-preserving and there is a natural transformation $\eta : F' \to F$ and for all relations R and all $(P_1, P_2) \in F'(R)$ we find $\eta(P_1)\, F(R)\, \eta(P_2)$. Then, the relation $\mathcal{T}$ is also $(F', \eta(H_1), \eta(H_2))$-preserving where $\eta(H) = \{x \in F'(ClosedSys) \mid \eta_{ClosedSys}(x) \in H\}$.*

If the relation $\mathcal{T}$ is just a function, then a natural transformation between functors is enough and the extra condition is not necessary.

An example of such an implication is $RrHP \implies RKrHP$ from [6]. That is preservation of ∞-ary properties implies preservation of k-ary properties.

There are other interesting arities that still warrant more detailed study, such as a reduction of indistinguishability games implying full abstraction or vice versa, since security games often model indistinguishability as a unary probabilistic property rather than a binary one by making a program flip a coin to choose how to behave.

5 EXAMPLES OF SYSTEM MODELS AND ROBUST PROPERTY PRESERVATION

To help the reader understand system models, we discuss how to model some of our guiding examples as system models and discuss how robust properties appear in these cases. We also show in these cases that the formulation of (F, H_1, H_2)-preserving relations can succinctly express several results about the preservation of robust properties.

5.1 Computability theory

As mentioned in Section 2.1, we consider a set of strings A called a language and the contexts are Turing machines. Putting this into a system model we can study $SysModel_A = SysModel(\{A\}, TM, \bowtie)$ where TM denotes the set of all Turing machines and $\bowtie$ links the language with the Turing machine. Now, we discussed that undecidability is a robust property, specifically, no Turing machine linked with the language can decide the language. So, the undecidability of language A can be expressed as $A \in robustly(id, H)$ in $SysModel_A$ with $H_{\neg decides} = \{Cs \mid Cs = M \bowtie A' \wedge \neg decides(M \bowtie A')\}$. Here, $decides(M \bowtie A)$ is true if Turing machine M decides language A and false otherwise. Here, id is the identity functor which indicates that we only consider properties on one program at a time. Note,

that in this case, we are only studying one language per system model.

In computability theory, we find reduction proofs that show that if A is undecidable, then B is undecidable. This is exactly the fact that $\mathcal{T} : SysModel_A \to SysModel_B : A \mapsto B$ is a $(id, H_{\neg decides}, H_{\neg decides})$-preserving relation. Note that in this case, the C_1 in the back-translation version of our criterion corresponds to the decider for language A that is constructed in the reduction proof given the existence of a decider for language B.

5.2 Complexity theory

The example of complexity theory is very similar to the computability theory except that in this case the set of deciders is restricted to the set of Turing machines whose runtime is a certain function of the length of their input (e.g. polynomial). So, the system model for language A is $SysModel_{A,\text{poly}} = SysModel(\{A\}, \text{TM-P}, \bowtie)$ and language A is not an element of P is exactly the same robust property as in Section 5.1, namely $A \in robustly(id, H)$, but for $SysModel_{A,\text{poly}}$ instead of for $SysModel_A$.

Another approach could also be that the set of deciders is not restricted in the system model but in the security property. In this case, P^C [3] is equal to

$$robustly(id, \{Cs \mid Cs = M \bowtie A' \wedge \neg decides(M \bowtie A') \text{ or } M \notin \text{TM-P}\})$$

in $SysModel_A$.

Just, as in computability theory, the reduction proofs of complexity theory behave exactly the same as in computability theory, so $\mathcal{T} : SysModel_{A,\text{poly}} \to SysModel_{B,\text{poly}} : A \mapsto B$ is also a $(id, H_{\neg decides}, H_{\neg decides})$-preserving relation.

5.3 Cryptographic games

As mentioned in Section 2.3, cryptographic games are a challenge for an adversary parametric in a security parameter. Both the challenge and the adversary should be polynomial in this security parameter. Using this information, we get

$$SysModel_G = SysModel(\{G\}, \text{adversaries}_{\text{poly}}, \bowtie)$$

where G denotes the game and $\text{adversaries}_{\text{poly}}$ denotes the set of adversaries polynomial in the security parameter. Note here that the arity of these games is sequences of probability distributions, because games are probabilistic and indexed by the security parameter. The fact that a game cannot be won with a significant probability is a robust property, namely $G \in robustly(id, H_{\neg wins})$ where $wins(A \bowtie G') = \neg negligible(\mathbb{P}(A \bowtie G' = 1))$ and $H_{\neg wins} = \{Cs \mid Cs = A \bowtie G' \wedge \neg wins(A \bowtie G')\}$. Negligible in this case means that the probability decreases faster as a function of the security parameter than any inverse polynomial.

Now, we notice that reduction proofs between security games or even reductions to established hard problems are proofs that the compilers that replace one game by another preserve unwinnability. So, for example if game G_1 reduces to game G_2, then the compiler $\mathcal{T} : SysModel_{G_1} \to SysModel_{G_2} : G_1 \mapsto G_2$ is a $(id, H_{\neg wins}, H_{\neg wins})$-preserving compiler with $H_{\neg wins}$ as above.

[3] P^C indicates the complement of P, meaning the class languages not decidable by polynomial-time Turing machines

Note that in the case of indistinguishability games the probability should be close to $\frac{1}{2}$ instead of 0. In this case, *wins* would be defined differently.

5.4 Trace models

In the case of trace models, the set of programs are the partial programs of the programming language we are studying and the contexts are programs with holes. The linker fills the hole in the context by the program and applies the behaviour functionality to it, so we look only at the outputted trace. The appropriate system model here is $SysModel_{trace} = SysModel(Prog, Context, \bowtie)$ where $Prog$ is the set of programs in the language under consideration, $Context$ are the programs with holes (or any other type of context in a trace model) that are considered and $\bowtie$ is the outputs the resulting behaviour of plugging the program into the context.

In this section, contextual equivalence is a robust property, specifically, for any choice of equivalence $\equiv$ for your traces, contextual is $robustly(F : X \to X \times X, \equiv)$ where F is the functor that maps sets to sets of pairs. Now, full abstraction is the preservation of this contextual equivalence, so a compiler $[\![\cdot]\!]$ is fully abstract if and only if $[\![\cdot]\!]$ is $(F : X \to X \times X, \equiv_S, \equiv_T)$-preserving where $\equiv_S$ and $\equiv_T$ are the respective equivalence on the source and target traces.

The work by Abate et al. [6] has plenty of secure compilation criteria for the trace model that fit this framework. Now, note that a compiler is RHP if and only if he is (id, ID)-preserving where ID is the identity relation on hyperproperties about sets of traces. It is important here that the source and target system model use the same traces. Similarly, RrHP is $(F' : X \to X^{\mathbb{N}}, ID)$-preserving where ID is the identity relation on relational hyperproperties about sequences of sets of traces and F' is the functor that maps to sequences.

5.5 Computational soundness of cryptography

Now, we discuss computational soundness of cryptography [3] which is an instance of a compiler between two very different systems, namely from a deterministic system to a probabilistic system.

The deterministic system is the basic language presented in Section 2.5 and there is no context involved. We model no context as having only the trivial context $\{\cdot\}$, such that $\cdot \bowtie M = M$. The system model we discuss is $SysModel_{symb} = SysModel(Term_{symb}, \{\cdot\}, \bowtie)$ where $Term_{symb}$ denotes the set of terms in the symbolic language. Note that all properties are robust in this setting because the set of contexts is trivial.

The computational probabilistic system is more complicated. Let us call the set of considered terms, $\mathcal{L}_\Pi$, referring to the encryption triple Π. The set of considered adversaries are all probabilistic polynomial-time adversaries (polynomial in the security parameter also handed to $\mathcal{K}$) that receive a value sampled from the probability distribution of terms generated by the given coins and output 0 or 1 based on this. Specifically, these are machines that are polynomial-time and cannot win the Type-0 security game for Π. We define the system as $SysModel_\Pi = SysModel(\mathcal{L}_\Pi, \text{adversaries}_\Pi, \bowtie)$ where $\bowtie$ just considers the pairs and adversaries_Π are the polynomial-time adversaries that lose the Type-0 security challenge for Π. Indistinguishability in $SysModel_\Pi$ is a robust property and can be defined

as

$$robustly(X \to X \times X, \\ \{(Cs_1, Cs_2) \mid \neg negligibleDiff(Cs_1, Cs_2)\})$$

where $negligibleDiff(A \bowtie M, A' \bowtie N) = negligible(\mathbb{P}(A \bowtie M = 1) - \mathbb{P}(A' \bowtie N = 1))$.

Abadi and Rogaway [3] showed that the computational encryption function is sound by constructing a compiler from $SysModel_{symb}$ to $SysModel_{\Pi}$. Specifically, they showed that their compiler was fully abstract with respect two different equivalences in the source and target. In the source the equivalence was defined by having the same pattern as explained in the second paragraph of Section 2.5. The target equivalence is the indistinguishability explained above. So, defined in our framework the compiler is

$$(X \to X \times X, samepatterns, \\ \{(Cs_1, Cs_2) \mid \neg negligibleDiff(Cs_1, Cs_2)\})$$

where same patterns is the set of pairs of terms that result in the same pattern.

5.6 Layout randomisation

In Section 2.6, we presented the work by Abadi and Plotkin on layout randomisation [2] which features an interesting compiler from a deterministic language to a probabilistic language.

The high-level language and its contexts form a system model $SysModel_{\text{high}} = \bigsqcup_\sigma SysModel(\mathcal{P}^\sigma_{\text{high}}, C^\sigma_{\text{high}}, \bowtie)$ where $\mathcal{P}^\sigma_{\text{high}}$ are the programs of type σ in the high-level language and C^σ_{high} are the contexts with a hole of type σ in the high-level language. Here, they define contextual equivalence which is a robust binary property.

The low-level language is very similar except that the link of a program and its context is not actually a whole program, but a function from memory layouts to whole programs. Another point of view is that they are distributions of whole programs where the uniform distribution over memory layouts is already taken into account. So, the system model is

$$SysModel_{\text{low}} = \bigsqcup_\sigma SysModel(\mathcal{P}^\sigma_{\text{low}}, C^\sigma_{\text{low}}, \bowtie)$$

where $\mathcal{P}^\sigma_{\text{low}}$ are the programs of type σ in the low-level language and C^σ_{low} are the context with a hole of type σ in the low-level language. Two equivalent programs at low-level either have the same output (evaluate to the same value or both diverge) or they both fail with a sufficiently high probability. We indicate equivalent programs in the low-level language with $M \sim N$. So, contextually equivalent up to fatal errors is $robustly(X \to X \times X, \sim)$.

The compiler in [2] is also fully abstract with the equivalence relations as discussed in Section 5.6. However, something more interesting is happening. They prove full abstraction by back-translation and their proof for the back-translation actually proves that the compiler is $(F' : X \to X^{\mathbb{N}}, R)$-preserving with R the following relation: $R = \{(\{H_i\}_{i\in\mathbb{N}}, F'(f)(\{H_i\}_{i\in\mathbb{N}}))\}$ where f is the relation that relates a high-level program to all low-level programs that have the same execution or if the high-level program fails, then it relates to all low-level programs that fail with sufficient probability. This is a more powerful criterion for our compiler. Another interesting thing is that we can still easily derive full abstraction from this. First, we construct the natural transformation $\eta : F \to F' : (X_1, X_2) \to (X_1, X_2, X_1, X_2, \ldots)$ where $F : X \to X \times X$ is the pair functor and $F' : X \to X^{\mathbb{N}}$ is the sequence functor. Invoking Theorem 1 on the preservation result, we get that the compiler is $(F, \{((H_1, H_2), F(f)(H_1, H_2))\}$-preserving. This is the preservation of binary properties up to only high probability of fatal error in the target (instead 100% probability). By noticing that equivalence in the target is actually $F(f)$ applied to equivalence in the source, we obtain the full abstraction result from [2].

5.7 Universal composability (UC)

In universal composability (UC) (c.f. Section 2.7), we discuss the UC-emulation of protocols. The protocols are interactive Turing machines that are polynomial in the security parameter k in the way discussed in Section 2.7 (polynomial in the import and needing at least k import to start). The same holds for the adversaries that are polynomial and can only communicate with protocols through the backdoor tapes and with the designated environment. So, in this case the system is $SysModel_\pi = SysModel(\{\pi\}, \text{UC-adversaries}, \bowtie)$ where link means putting the protocol and adversary together as a set of ITMs. There are not really any robust properties discussed in the work around UC, but the link with robust properties and secure compilation will quickly become clear once we discuss preservation of robust properties.

Note, that besides UC-emulation of specific protocols, we also have the universal composition theorem which talks about the mapping $.^{\phi\to\pi}$. In this case, the system of interest is $SysModel_{ITM_{\text{poly}}} = SysModel(ITM_{\text{poly}}, \text{UC-adversaries}, \bowtie)$ where ITM_{poly} is the set of ITMs that run in polynomial time in the security parameter and can be dependent on the security parameter in a principled way.

In UC, we have the notion of protocol π UC-emulating ϕ as explained in Section 2.7. We note that this notion corresponds exactly with the backtranslation version of (id, R)-preserving the relation between ϕ and π where the back-translated context is the simulator. However, UC-emulation then says that $A \bowtie \pi$ is indistinguishable from $S \bowtie \phi$. We express this indistinguishability in our framework by saying that the compiler does not preserve one property but it preserve each robust property H that respects indistinguishability. Respecting indistinguishability means that if $M \in H$ then for each N that is indistinguishable from M the statement $N \in H$ has to hold. So, π UC-emulates ϕ means that the relation from $SysModel_{phi}$ to $SysModel_\pi$ which only relates ϕ to π is $(id, Id/Indist)$-preserving where $Id/Indist$ is the identity relation, but only for properties on ITMs that respect indistinguishability.

Remember that the UC framework also discusses the universal composition theorem which talks about the $.^{\phi\to\pi}$ operation. Specifically, it says that if π UC-emulates ϕ, then for all ITMs ρ that use ϕ as subroutine, we find that ρ UC-emulates $\rho^{\phi\to\pi}$ where $.^{\phi\to\pi}$ is the operation that replaces all calls to ϕ by calls to π. This means that the compiler $.^{\phi\to\pi} : SysModel_{UC} \to SysModel_{UC}$ is $(id, Id/Indist)$-preserving.

6 RELATED WORK

This related work section is quite short, because most related work has already been discussed in detail throughout this paper.

The idea of robust properties was first introduced by Grumberg and Long [11]. They used this idea to check parts of a temporal logic model separately and modularly. Subsequently, Kupferman and Vardi [12] seem to be the first to call it robust satisfaction of a property. They used this term in the area of reactive systems. Specifically, "Given an open system M and a property ψ, we say that M robustly satisfies ψ iff for every open system M', which serves as an environment to M, the composition $M||M'$ satisfies ψ." Their work then focuses more on checking whether a model robustly satisfies the property.

The field of secure compilation also discusses robust properties and their preservation in depth. In his seminal paper, Abadi [1] suggested to characterise preservation of protection in programming-language translations by the criterion of full abstraction which is the preservation of contextual equivalence, which is a robust property. However, there are certain desirable properties about preservation of security that the criterion of full abstraction does not deliver and the proofs used for full abstraction often prove more powerful results than just full abstraction. For these reasons Abate et al. [6] proposed a family of new criteria for secure compilers based on the idea of preserving trace properties/hyperproperties/relational hyperproperties robustly. Robustly means here that the property is satisfied under linking with any context and trace properties/hyperproperties/relational hyperproperties are respectively sets of allowed traces/properties of sets of possible traces of a single program/relations between the sets of possible traces of two or more programs. Furthermore, they look at secure compilation only in a specific setting of programming languages with traces and contexts that are programs with holes. This view works quite well in a lot of cases, but there are works such as [2] which use probabilistic output and this does not fit into the framework.

One possible shortcoming of the work by Abate et al. [6] is the fact that the source and target language of a secure compiler need to have the same trace semantics. However, in a follow-up paper [5], Abate et al. devise new criteria for which it is sufficient to only relate the trace semantics by some relation.

7 CONCLUSION AND FUTURE WORK

In this paper, we introduced a general framework that describes several existing secure compilation criteria and theorems that are proved by reduction. We suspect that some results of full abstraction in the literature can easily be made more powerful by putting the back-translation result into this framework and looking at the associated property preservation (see e.g. our treatment of [2]).

Furthermore, we hope to look at aspects in this framework that one discipline uses and some others do not, to formulate and prove new preservation criteria. For example, relational hyperproperties in UC.

We also plan to apply this framework to the field of domain theory and language expressiveness. For example, in language expressiveness they already determined that full abstraction is insufficient to measure expressiveness [10].

Finally, we also plan to flesh out the framework more to get some more interesting overarching results.

ACKNOWLEDGEMENTS

This work was partially supported by the Research Fund KU Leuven, and by the Flemish Research Programme Cybersecurity, by the Air Force Office of Scientific Research under award number FA9550-21-1-0054 and by the Research Foundation - Flanders (Research Project G030320N).

REFERENCES

[1] Martín Abadi. 1999. Protection in Programming-Language Translations. In Secure Internet Programming. Lecture Notes in Computer Science, Vol. 1603. Springer Berlin Heidelberg, Berlin, Heidelberg, 19–34.

[2] Martín Abadi and Gordon D. Plotkin. 2012. On Protection by Layout Randomization. ACM Trans. Inf. Syst. Secur. 15, 2, Article 8 (jul 2012), 29 pages. https://doi.org/10.1145/2240276.2240279

[3] Martín Abadi and Phillip Rogaway. 2002. Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)*. Journal of Cryptology 15, 2 (Jan. 2002), 103–127. https://doi.org/10.1007/s00145-001-0014-7

[4] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2021. An Extended Account of Trace-Relating Compiler Correctness and Secure Compilation. ACM Trans. Program. Lang. Syst. 43, 4, Article 14 (nov 2021), 48 pages. https://doi.org/10.1145/3460860

[5] Carmine Abate, Roberto Blanco, Ştefan Ciobâcă, Adrien Durier, Deepak Garg, Cătălin Hriţcu, Marco Patrignani, Éric Tanter, and Jérémy Thibault. 2020. Trace-Relating Compiler Correctness and Secure Compilation. In Programming Languages and Systems, Peter Müller (Ed.). Springer International Publishing, Cham, 1–28.

[6] Carmine Abate, Roberto Blanco, Deepak Garg, Catalin Hritcu, Marco Patrignani, and Jérémy Thibault. 2019. Journey Beyond Full Abstraction: Exploring Robust Property Preservation for Secure Compilation. In 2019 IEEE 32nd Computer Security Foundations Symposium (CSF). 256–25615. https://doi.org/10.1109/CSF.2019.00025

[7] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. 2014. Easycrypt: A tutorial. Foundations of Security Analysis and Design VII: FOSAD 2012/2013 Tutorial Lectures (2014), 146–166.

[8] R. Canetti. 2001. Universally composable security: a new paradigm for cryptographic protocols. In Proceedings 42nd IEEE Symposium on Foundations of Computer Science. 136–145. https://doi.org/10.1109/SFCS.2001.959888

[9] Ran Canetti. 2020. Universally Composable Security. J. ACM 67, 5, Article 28 (sep 2020), 94 pages. https://doi.org/10.1145/3402457

[10] DANIELE GORLA and UWE NESTMANN. 2016. Full abstraction for expressiveness: history, myths and facts. Mathematical Structures in Computer Science 26, 4 (2016), 639–654. https://doi.org/10.1017/S0960129514000279

[11] Orna Grumberg and David E. Long. 1994. Model Checking and Modular Verification. ACM Trans. Program. Lang. Syst. 16, 3 (may 1994), 843–871. https://doi.org/10.1145/177492.177725

[12] Orna Kupferman and Moshe Y. Vardi. 1999. Robust Satisfaction. In CONCUR'99 Concurrency Theory, Jos C. M. Baeten and Sjouke Mauw (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 383–398.

[13] Marco Patrignani, Riad S. Wahby, and Robert Künneman. 2019. Universal Composability is Robust Compilation. CoRR abs/1910.08634 (2019). arXiv:1910.08634 http://arxiv.org/abs/1910.08634

[14] nLab 2023. 2-category equipped with proarrows in nLab. nLab. https://ncatlab.org/nlab/show/2-category+equipped+with+proarrows

[15] Michael Sipser. 1997. Introduction to the theory of computation. International Thomson Publishing Ltd., London.

[16] Nigel Smart. 2016. Cryptography Made Simple (1st ed. 2016. ed.). Springer International Publishing : Imprint: Springer, Cham.

[17] Stelios Tsampas, Andreas Nuyts, Dominique Devriese, and Frank Piessens. 2020. A Categorical Approach to Secure Compilation. In Coalgebraic Methods in Computer Science, Daniela Petrişan and Jurriaan Rot (Eds.). Springer International Publishing, Cham, 155–179.