Computational Soundness for Dalvik Bytecode

Michael Backes CISPA, Saarland University & MPI-SWS Saarland Informatics Campus backes@cs.uni-saarland.de

Robert Künnemann CISPA, Saarland University Saarland Informatics Campus robert.kuennemann @uni-saarland.de

Esfandiar Mohammadi ETH Zurich mohammadi@inf.ethz.ch

ABSTRACT

Automatically analyzing information flow within Android applications that rely on cryptographic operations with their computational security guarantees imposes formidable challenges that existing approaches for understanding an app's behavior struggle to meet. These approaches do not distinguish cryptographic and non-cryptographic operations, and hence do not account for cryptographic protections: $f(m)$ is considered sensitive for a sensitive message m irrespective of potential secrecy properties offered by a cryptographic operation f. These approaches consequently provide a safe approximation of the app's behavior, but they mistakenly classify a large fraction of apps as potentially insecure and consequently yield overly pessimistic results.

In this paper, we show how cryptographic operations can be faithfully included into existing approaches for automated app analysis. To this end, we first show how cryptographic operations can be expressed as symbolic abstractions within the comprehensive Dalvik bytecode language. These abstractions are accessible to automated analysis and can be conveniently added to existing app analysis tools using minor changes in their semantics. Second, we show that our abstractions are faithful by providing the first computational soundness result for Dalvik bytecode, i.e., the absence of attacks against our symbolically abstracted program entails the absence of any attacks against a suitable cryptographic program realization. We cast our computational soundness result in the CoSP framework, which makes the result modular and composable.

Keywords

Android, Computational Soundness, Secure Information Flow

1. INTRODUCTION

Android constitutes an open-source project not only in terms of source code but also in terms of the whole ecosystem, allowing practically everyone to program new apps and make them publicly available in Google Play. This open nature of Android has facilitated a rapid pace of innovation, but it has also led to the creation and widespread deployment of malicious apps [1, 2]. Such apps often cause privacy violations that leak sensitive information such as location or the user's address book, either as an intended functionality or as a result of uninformed programming. In some cases such apps can even extract sensitive information from honest apps.

A comprehensive line of research has, hence, strived to rigorously analyze how apps are accessing and processing sensitive information. These approaches typically employ the concept of information flow control (IFC), i.e., certain information sources such as GPS position and address book are declared to be sensitive, and certain information sinks are declared to be adversarially observable. An IFC-based analysis then traces the propagation of sensitive information through the program, i.e., if sensitive data m is input to a function f, then the result $f(m)$ is considered sensitive as well. IFC-based analyses thereby determine if information from sensitive sources can ever reach an observable sink, and in that case report a privacy violation.

A considerable number of apps rely on cryptographic operations, e.g., for encrypting sensitive information before it is sent over the Internet. However, analyzing information flow within Android apps that rely on such cryptographic operations with their computational security guarantees imposes formidable challenges that all existing approaches for automated app analysis struggle to meet, e.g., [3–5]. Roughly, these approaches do not distinguish cryptographic operations from other, non-cryptographic functions. Thus, the standard information-tracing mechanism for arbitrary functions applies: $f(m)$ is considered sensitive for a sensitive message m irrespective of potential secrecy properties offered by a cryptographic function f, e.g., the encryption of a sensitive message m is still considered sensitive such that sending this encryption over the Internet is considered a privacy breach. These approaches consequently provide a safe approximation of the app's behavior, but they mistakenly classify a large fraction of apps as potentially insecure and consequently yield *overly pessimistic results*. While approaches based on manual declassification have successfully managed to treat cryptographic operations and their protective properties more accurately, see the section on related work for more details, no concept for an accurate cryptographic treatment is known for automated analysis of Android apps.

CCS'16 October 24–28, 2016, Vienna, Austria

DOI: http://dx.doi.org/10.1145/2976749.2978418

1.1 Our Contributions

In this paper, we show how cryptographic operations can be faithfully included into existing approaches for automated app analysis on Android, in the presence of malicious apps or network parties aiming to extract sensitive information from honest parties. Our paper makes two main tangible contributions to this field: (i) we show how cryptographic operations can be expressed as symbolic abstractions within Dalvik bytecode, so that existing automated analysis tools can adopt them with only minor changes in their semantics; and (ii) we show that our abstractions are faithful by providing the first computational soundness result for the comprehensive Dalvik bytecode language, i.e., the absence of attacks against our symbolically abstracted program entails the absence of any attacks against a suitable cryptographic program realization.

Symbolic abstractions in Dalvik bytecode. We first show how cryptographic operations can be expressed as symbolic abstractions within Dalvik bytecode. These symbolic abstractions – often also referred to as perfect cryptography or Dolev-Yao models – constitute idealizations of cryptographic operations as free algebras that lend themselves towards automated analysis. Deriving such abstractions within the comprehensive Dalvik bytecode language constitutes a challenging task, since existing formalizations of Dalvik do not offer a distinction between honest and adversarially controlled components, which is crucial for defining the rules that the symbolic adversary has to adhere to. To this end, we develop a novel semantic characterization of Dalvik bytecode that we call *split-state semantics* that provides a clear separation between honest program parts and cryptographic API calls with their corresponding augmented adversarial symbolic capabilities. Moreover, this split-state form is key to our proof of computational soundness, see below. Existing tools for automated app analysis can conveniently include our abstractions using minor changes in their underlying semantics, and thereby reason more accurately about cryptographic operations.

Computational soundness for Dalvik bytecode. We show that our symbolic abstractions can be securely instantiated using suitable cryptographic primitives, and thereby provide the first computational soundness result for Dalvik bytecode. More specifically, our result is grounded in the Abstract Dalvik Language (ADL) [4], which constitutes the currently most detailed and comprehensive operational semantics for Dalvik in the literature. To this end, we first extended ADL by probabilistic choices, as it otherwise would be inappropriate to express cryptographic operations.

We cast our computational soundness result in CoSP, a framework for establishing computational soundness results that decouples the process of embedding programming languages into CoSP from the computational soundness proofs itself. In particular, by casting our soundness results in CoSP, a rich body of computational soundness results for individual cryptographic primitves [6–11] is immediately valid for Dalvik bytecode without any additional work.

Establishing computational soundness results for Dalvik bytecode imposed a series of technical challenges that many prior computational soundness works did not have to cope with. We highlight one such challenge. Computational soundness results struggle when confronted with situations in which binary operations are applied to outputs of cryptographic operations, e.g., if the parity of a ciphertext should be checked, since such an operation would be undefined in the symbolic setting. Prior computational soundness results simply excluded programs with such illegitimate operations; this exclusion introduced an additional proof obligation for the automated analysis tool. While excluding such programs simplifies the soundness result, integrating these additional proof obligations in existing automated app analysis tools constitutes a tedious task, since the tools would need to check upfront whether any symbolically undefined operations will be performed on symbolic terms, in any execution branch. As a consequence, we hence decided to establish a computational soundness result that over-approximates such scenarios by sending information of such illegitimate operations to the adversary and letting the adversary decide the result of such operations.

Finally, our proof reveals an additional result that we consider of independent interest: we show that any small-step semantics S in split-state form entails a canonical small-step semantics S^* for a symbolic model that is computationally sound with respect to S. Hence, for establishing a computationally sound symbolic abstraction for *any* given programming language, it suffices to show that the interaction with the attacker and the cryptographic API can be expressed by means of our concept of split-state semantics.

1.2 Summary of Our Techniques

This section summarizes the techniques that we use to achieve these results. We hope that this summary makes the paper better accessible, given that it was distilled from a technical report spanning over roughly 50 pages [12]. Finally, we discuss how our results can be used to extend information flow tools.

The CoSP framework. A central idea of our work is to reduce computational soundness of Dalvik bytecode to computational soundness in the CoSP framework [6,10]. All definitions in CoSP are cast relative to a *symbolic model* that specifies a set of constructors and destructors that symbolically represent the cryptographic operations and are also used for characterizing the terms that the attacker can derive (called *symbolic attacker knowledge*), and a *computational implementation* that specifies cryptographic algorithms for these constructors and destructors. In CoSP, a *protocol* is represented by an infinite tree that describes the protocol as a labeled transition system. Such a CoSP protocol contains actions for performing abstract computations (applying constructors and destructors to messages) and for communicating with an adversary. A CoSP protocol is equipped with two different semantics: (i) a *symbolic CoSP execution*, in which messages are represented by terms; and (ii) a *computational CoSP execution*, in which messages are bitstrings, and the computational implementation is used instead of applying constructors and destructors. A computational implementation is said to be *computationally sound* for a class of security properties if any CoSP protocol that satisfies these properties in the symbolic execution also satisfies these properties in the computational execution. The advantage of expressing computational soundness results in CoSP is that the protocol model in CoSP is very general so that the semantics of other languages can be embedded therein, thereby transferring the various established soundness results from CoSP into these languages [6–11].

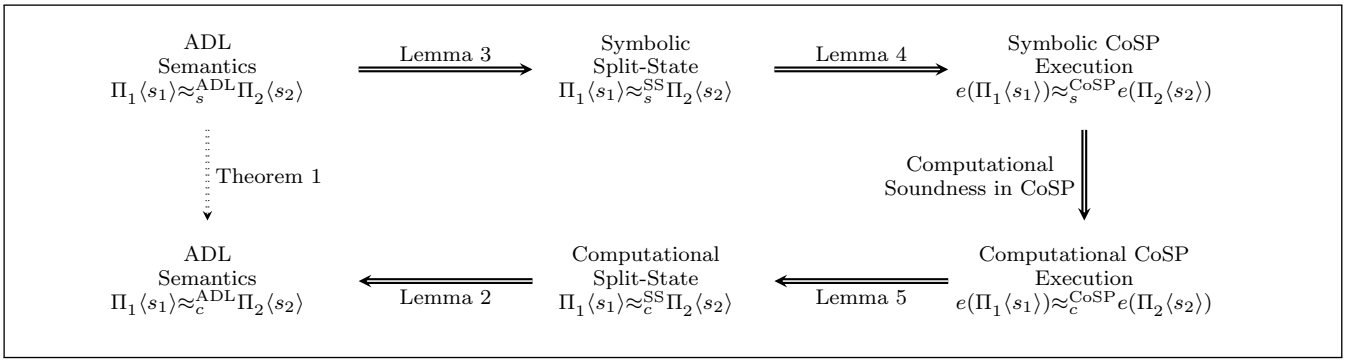


Figure 1: Overview of the main technical lemmas, where e is the embedding into CoSP

Symbolic ADL and probabilistic choices. ADL as defined in [4] does not support probabilistic choices, and hence no generation of cryptographic keys and no executions of cryptographic functions. We thus extended ADL with a rule that uniformly samples random values (more concretely: a register value from the set of numerical values).

We consider attackers that are external to the app, e.g., malicious parties or network parties with the goal of extracting secrets from an honest app. We characterize the interaction points of the attacker with our extended version of ADL by a set of so-called *malicious functions* that communicate with the attacker. The attacker itself is modeled as a probabilistic polynomial-time machine. We introduce an additional semantic rule that is applied whenever a malicious function is called. The rule invokes the attacker with the arguments of the function call and stores the response of the attacker as a return value. For defining the indistinguishability of two ADL programs, we additionally require that the adversary can also send a single bit b as a final guess, similar to other indistinguishability definitions. This entails the notions of *symbolic equivalence* ($\approx_s^{\mathrm{ADL}}$, in the symbolic setting) and of *computational indistinguishability* ($\approx_c^{\mathrm{ADL}}$, in the computational setting) of two ADL programs.

Split-state semantics for symbolic ADL. Establishing a computational soundness proof for ADL requires a clear separation between honest program parts and cryptographic API calls with their corresponding augmented adversarial symbolic capabilities. To achieve this, we characterize this partitioning by introducing the concept of a *split-state form* of an operational semantics. The split-state form partitions the original semantics into three components, parallelly executed asynchronously: (*i*) all steps that belong to computing cryptographic operations (called the *crypto-API semantics*), (*ii*) all steps that belong to computing the malicious functions (called the *attacker semantics*), and (*iii*) all steps that belong to the rest of the program (called the *honest-program semantics*). Moreover, we define explicit transitions between each of these components, which gives rise to a precise message-passing interface for cryptographic operations and for communicating with the attacker.

Our strategy for showing computational soundness is to use this split-state form for phrasing the symbolic variant as a small-step semantics by replacing the crypto-API semantics with the symbolic constructors and destructors from the symbolic model, and by replacing the attacker semantics by the symbolic characterization of the attacker. With this symbolic semantics at hand, we define split-state symbolic equivalence ($\approx_s^{\mathrm{SS}}$) as equivalence of (sets of) traces. However, as explained before, we first have to resolve the problem that computational soundness results struggle to deal with situations in which binary operations are applied to outputs of cryptographic operations. We decided not to exclude programs that exhibit such behaviors, but to perform an over-approximation instead by letting the adversary determine the outcome of such operations. This makes our abstractions conveniently accessible for existing tools, but it also complicates the computational soundness proof since we have to consider the operations of constructors and destructors on non-symbolic terms as well. To this end, we encode them as bitstrings and interpret these bitstrings symbolically again. Fortunately, symbolic bitstring interpretations can be seamlessly combined with all previous CoSP results.

We finally define computational indistinguishability of two honest program semantics in the split-state computational execution of ADL ($\approx_c^{\mathrm{SS}}$). As usual, the adversary we consider is a probabilistic polynomial-time machine, and all semantics constitute families of semantics that are indexed by a security parameter.

It might be of independent interest that our symbolic variant of the semantics and the computational indistinguishability can be defined on the split-state form independently of ADL. We show that ADL can be brought into such a split-state form, then prove later that symbolic equivalence in ADL implies symbolic equivalence in the split-state form, and conclude by proving that computational indistinguishability in the split-state form implies computational indistinguishability in ADL. Hence, for every two ADL programs Π_1, Π_2 and initial configurations s_1, s_2 ($\Pi_i\langle s_i\rangle$ denoting Π_i with initial configuration s_i) we have

$$\Pi_1\langle s_1\rangle \approx_s^{\mathrm{ADL}} \Pi_2\langle s_2\rangle \implies \Pi_1\langle s_1\rangle \approx_s^{\mathrm{SS}} \Pi_2\langle s_2\rangle, \text{ and}$$
$$\Pi_1\langle s_1\rangle \approx_c^{\mathrm{SS}} \Pi_2\langle s_2\rangle \implies \Pi_1\langle s_1\rangle \approx_c^{\mathrm{ADL}} \Pi_2\langle s_2\rangle.$$

Computational soundness proof. We first construct an injective embedding e that maps every ADL program to a CoSP protocol. We stress that within CoSP, the same CoSP protocol is used for the computational and the symbolic execution and that CoSP requires a separation of the attacker and the cryptographic operations from the rest of the program. Our split-state form precisely satisfies these requirements. The embedding e uses the honest-program semantics to iteratively construct a CoSP protocol: a transition to the crypto-API semantics corresponds to a computation node; a

transition to the attacker semantics corresponds to an output node followed by an input node; and whenever several possibilities exist, a control node is selected to let the adversary decide which possibility (which node) to take.

We prove this embedding sound in the symbolic model, and we prove it complete with respect to the range of e in the computational model, i.e., for every two ADL programs Π_1, Π_2 and initial configurations s_1, s_2 we have

$$\Pi_1\langle s_1\rangle \approx_s^{\mathrm{SS}} \Pi_2\langle s_2\rangle \implies e(\Pi_1\langle s_1\rangle) \approx_s^{\mathrm{CoSP}} e(\Pi_2\langle s_2\rangle)$$

and

$$e(\Pi_1\langle s_1\rangle) \approx_c^{\mathrm{CoSP}} e(\Pi_2\langle s_2\rangle) \implies \Pi_1\langle s_1\rangle \approx_c^{\mathrm{SS}} \Pi_2\langle s_2\rangle,$$

where $\approx_s^{\mathrm{CoSP}}$ and $\approx_c^{\mathrm{CoSP}}$ denote symbolic equivalence and computational indistinguishability in CoSP.

Figure 1 finally shows how all pieces are put together:

Theorem 1 (computational soundness of Dalvik – simplified). *Let Π_1, Π_2 be two ADL programs that use the same crypto-API and s_1, s_2 be two initial configurations. Then we have*

$$\Pi_1\langle s_1\rangle \approx_s^{\mathrm{ADL}} \Pi_2\langle s_2\rangle \implies \Pi_1\langle s_1\rangle \approx_c^{\mathrm{ADL}} \Pi_2\langle s_2\rangle.$$

Extension of information flow tools. To put our work in perspective, we elaborate on a possible application of our result. We envision the extension of information flow (IF) methods with symbolic abstractions. On a high-level, we envision the following approach for extending IF-tools: whenever there is a potential information flow from High to Low and a cryptographic function is called, we extract a model of the app and query a symbolic prover to find our whether the attacker learns something about the High values. As in symbolic ADL only a few semantic rules are changed w.r.t. ADL, modifications to existing analyses are likely to be confined, thus simple to integrate.

This approach imposes the challenge of extracting a model of the app. Our embedding of an ADL program into CoSP already extracts a symbolic model for the program. However, shrinking this extracted model to a manageable size and querying the symbolic prover in a way such that it scales to complex apps is a task far from simple that merits a paper on its own.

1.3 Overview

Section 3 reviews the CoSP framework for equivalence properties that we ground our computational soundness result on. Section 4 reviews the Abstract Dalvik Language (ADL). Sections 5 and 6 define the probabilistic execution of ADL and introduce our symbolic variant of ADL, including the symbolic abstractions of cryptographic operations and the capabilities of the symbolic adversary. Section 7 defines the connections between ADL, symbolic ADL, and CoSP, and based on these connections proves the computational soundness result. Section 8 discusses related work. We conclude in Section 9 with a summary of our findings and outline directions for future research.

2. NOTATION

Let $\mathbb{N}$ be the set of natural numbers and assume that they begin at 0. For indicating that function f from a set A to a set B is a partial function, we write $f : A \rightharpoonup B$. We use squared brackets in two different ways: (*i*) $m[pp]$ denotes the instruction with the number pp for a set of instructions m, and (*ii*) $r[v \mapsto val] := (r \backslash (v, r(v))) \cup (v, val)$ is short-hand for the function mapping v to val and otherwise behaving like r. Throughout the paper, we use η as the security parameter. We use ε to denote the empty sequence, empty path, empty bitstring, or empty action depending on the context. We write $\underline{t}$ for a sequence $t_1, \ldots, t_n$ if n is clear from the context. For any sequence $l \in E^*$, we use $\cdot$ to denote concatenation $l_1 \cdot l_2$, as well as the result of appending $(l \cdot e)$ or prepending $(e \cdot l)$ an element, as long as the difference is clear from context. We filter a sequence l by a set S, denoted $l|_S$, by removing each element that is not in S. As we represent the attacker as a transition system, we sometimes write T_A and sometimes $\mathcal{A}$ for the attacker.

3. COSP FRAMEWORK (REVIEW)

The computational soundness proof developed in this paper follows CoSP [6,10], a general framework for conducting computational soundness proofs of symbolic cryptography and for embedding these proofs into programming languages with their given semantics. This section reviews the CoSP framework.

Symbolic Model & Execution. In CoSP, symbolic abstractions of protocols and the attacker are formulated in a *symbolic model* $\mathbf{M} = (\mathbf{C}, \mathbf{N}, \mathbf{T}, \mathbf{D})$: a set of free functions $\mathbf{C}$, a countably infinite set $\mathbf{N}$ of nonces, a set $\mathbf{T}$ of terms, and a set $\mathbf{D}$ of partial mappings from terms to terms (called destructors). To unify notation, we introduce $eval_F(t)$: if F is a constructor, $eval_F(t) := F(t)$ if $F(t) \in \mathbf{T}$ and $eval_F(t) := \bot$ otherwise. If F is a nonce, $eval_F() := F$. If F is a destructor, $eval_F(t) := F(t)$ if $F(t) \neq \bot$ and $eval_F(t) := \bot$ otherwise.

Protocols. In CoSP, protocols are represented as infinite trees with the following nodes: *computation nodes* are used for drawing fresh nonces and for applying constructors and destructors; *input* and *output nodes* are used for send and receive operations; *control nodes* are used for allowing the attacker to schedule the protocol. A computation node is annotated with its arguments and has two outgoing edges: a yes-edge, used for the application of constructors, for drawing a nonce, and for the successful application of a destructor, and a no-edge, used if an application of a constructor or destructor F on a term t fails, i.e., if $eval_F(t) = \bot$. Nodes have explicit *references* to other nodes whose terms they use.

Symbolic operations. To model the capabilities of the symbolic attacker, we explicitly list the tests and operations that the attacker can perform on protocol messages using so-called *symbolic operations*. A symbolic operation is (similar to a CoSP tree) a finite tree whose nodes are labeled with constructors, destructors, or nonces from the symbolic model $\mathbf{M}$, or formal parameters x_i denoting pointers to the ith protocol message. There is a natural evaluation function for a symbolic operation O and a list $\underline{t}$ of terms that the attacker received so far (the *view*).

Symbolic execution. A symbolic execution is a path through a protocol tree defined as defined below. It induces a *symbolic view*, which contains the communication with the attacker. We, moreover, define an *attacker strategy* as the sequence of symbolic operations that the attacker performs in the symbolic execution.

$$(V,\nu,f) \rightsquigarrow_{CoSPs} (V, yes(\nu), f[\nu \mapsto m]) \qquad \nu \text{ computation n. with } F \in \mathbf{C}\cup\mathbf{D}\cup\mathbf{N},\ m := eval_F(\underline{\tilde{t}}) \neq \bot$$
$$(V,\nu,f) \rightsquigarrow_{CoSPs} (V, no(\nu), f) \qquad \nu \text{ computation node with } F \in \mathbf{C}\cup\mathbf{D}\cup\mathbf{N},\ eval_F(\underline{\tilde{t}}) = \bot$$
$$(V,\nu,f) \rightsquigarrow_{CoSPs} (V \cdot (\mathtt{in},(t,O)), succ(\nu), f[\nu \mapsto t]) \qquad \nu \text{ input node, } t \in \mathbf{T},\ O \in \mathrm{SO}(\mathbf{M}),\ eval_O(Out(V)) = t$$
$$(V,\nu,f) \rightsquigarrow_{CoSPs} (V \cdot (\mathtt{out},\tilde{t}_1), succ(\nu), f) \qquad \nu \text{ output node}$$
$$(V,\nu,f) \rightsquigarrow_{CoSPs} (V \cdot (\mathtt{control},(l,l')), \nu', f) \qquad \nu \text{ control n., out-metadata } l, \text{ successor } \nu' \text{ has in-metadata } l'$$

Figure 2: Rules for defining the smallest relation for the symbolic execution

Definition 1 (Symbolic executions). *Let a symbolic model* $\mathbf{M} = (\mathbf{C},\mathbf{N},\mathbf{T},\mathbf{D})$ *and a CoSP protocol* Π *for* $\mathbf{M}$ *be given. Let* $Views = (Event_{in} \cup Event_{out} \cup Event_{ctl})^*$, *with* $Event_{in} := \{\mathtt{in}\} \times \mathbf{T} \times \mathrm{SO}(\mathbf{M})$, $Event_{out} := \{\mathtt{out}\} \times \mathbf{T}^*$, $Event_{ctl} := \{\mathtt{control}\} \times \{0,1\}^* \times \{0,1\}^*$. *We define*

$$\rightsquigarrow_{CoSPs} \subseteq Views^* \times Nodes \times (Nodes \to \mathbf{T}) \to Views^* \times Nodes \times (Nodes \to \mathbf{T})$$

as the smallest relation s.t. the rules from Figure 2 hold, where $\underline{\tilde{t}} := f(\tilde{\nu}_1),\ldots,f(\tilde{\nu}_{|\underline{\tilde{\nu}}|})$, *for the nodes* $\tilde{\nu}_1,\ldots,\tilde{\nu}_{|\underline{\tilde{\nu}}|}$ *referenced by* ν *(for computation nodes). The set of* symbolic executions of Π *is defined as*

$$\mathrm{SExec}(\Pi) := \{((V_0,\nu_0,f_0),\ldots,(V_n,\nu_n,f_n)) \mid n \in \mathbb{N} \wedge \forall i.(V_i,\nu_i,f_i) \rightsquigarrow_{CoSPs} (V_{i+1},\nu_{i+1},f_{i+1})\},$$

where $(V_0,\nu_0,f_0) = (\epsilon, root(\Pi), \emptyset)$. V_i *is called a* symbolic view *for step* i. *The set of symbolic views of* Π *is defined as*

$$\mathrm{SViews}(\Pi) := \{V_n \mid (V_0,\nu_0,f_0),\cdots,(V_n,\nu_n,f_n) \in \mathrm{SExec}(\Pi)\}$$

Given a view V, $Out(V)$ *denotes the list of terms* t *contained in* $(\mathtt{out},t) \in V$. *Out-Meta*(V) *denotes the list of terms* l *contained in elements of the form* $(\mathtt{control},(l,l'))$ *in the view* V. $In(V)$, *called the* attacker strategy, *denotes the list of terms that contains only entries of* V *of the form* $(\mathtt{in},(t,O))$ *or* $(\mathtt{control},(l,l'))$, *where for* $(\mathtt{in},(t,O))$ *the entry* $(\mathtt{in},O)$ *is stored and for* $(\mathtt{control},(l,l'))$ *the entry* $(\mathtt{control},l')$ *is stored.* $[In(V)]_{\mathrm{SViews}(\Pi)}$ *denotes the equivalence class of all views* $U \in \mathrm{SViews}(\Pi)$ *with* $In(U) = In(V)$.

Symbolic knowledge. The *symbolic knowledge* of the attacker comprises the results of all symbolic tests the attacker can perform on the messages output by the protocol. Given a view V with $|V_{Out}| = n$, we define the symbolic knowledge K_V as a function from symbolic operations on $\mathbf{M}$ to $\{\top,\bot\}$, where $\top$ comprises all results of $eval_O(V_{Out})$ that are different from $\bot$.

Symbolic Equivalence. Two views are *equivalent*, denoted as $V \sim V'$, if they (*i*) have the same structure (i.e., the same order of $\mathtt{out},\mathtt{in},\mathtt{control}$ entries), (*ii*) have the same out-metadata (i.e., $V_{Out\text{-}Meta} = V'_{Out\text{-}Meta}$), and (*iii*) lead to the same knowledge (i.e., $K_V = K_{V'}$). Finally, two CoSP protocols are *symbolically equivalent* ($\approx_s^{\mathrm{CoSP}}$) if its two variants lead to equivalent views when run with the same attacker.

Computational execution. In the computational setting, symbolic constructors and destructors are realized with cryptographic algorithms. A *computational implementation* is a family $\underline{\mathrm{Impl}} = (A_x)_{x\in\mathbf{C}\cup\mathbf{D}\cup\mathbf{N_P}}$ of deterministic polynomial-time algorithms Impl_F for each constructor or destructor $F \in \mathbf{C}\cup\mathbf{D}$ as well as a probabilistic polynomial-time (ppt) algorithm A_N for drawing protocol nonces $N \in \mathbf{N}$.

Computational execution. The *computational execution* of a protocol is a randomized interactive machine, called the *computational challenger*, that traverses the protocol tree and interacts with a ppt attacker $\mathcal{A}$: at a computation node the corresponding algorithm is run and depending on whether the algorithm succeeds or outputs $\bot$, either the yes-branch or the no-branch is taken; at an output node, the message is sent to the attacker, and at an input node a message is received by the attacker; at a control node the attacker sends a command that specifies which branch to take. The transcript of the execution contains the computational counterparts of a symbolic view.

Computational indistinguishability. We finally define *computational soundness* by requiring that symbolic equivalence implies computational indistinguishability.

Definition 2 (Computational Soundness). *An implementation* $\underline{\mathrm{Impl}}$ *of a given symbolic model* $\mathbf{M}$ *is* computationally sound *for* $\mathbf{M}$ *and a class* P *of efficient pairs of protocols if for every* $\Pi \in \mathrm{P}$, *we have that* Π *is computationally indistinguishable whenever* Π *is symbolically equivalent.*

4. DALVIK BYTECODE (REVIEW)

The Abstract Dalvik Language (ADL) [4] constitutes the currently most detailed and comprehensive operational semantics for Dalvik in the literature, even though it does not encompass concurrency and exceptions (as e.g. in [3]). We refer to Section 8 for further discussion on different Dalvik semantics. Due to lack of space, we only provide a very compact review of ADL and refer to [4] for more details.

Syntax. ADL methods are non-empty sequences of instructions and together constitute a set $\mathcal{M}$. A method name $mid \in \mathcal{MID}$ and a class name $c \in \mathcal{CID}$ identify the method to be called by means of lookup tables, i.e., partial functions mapping (c, mid) to normal methods ($lookup\text{-}direct_\Pi$), inherited methods ($lookup\text{-}super$), or virtual methods($lookup\text{-}virtual_\Pi$). In contrast, static methods ($lookup\text{-}static_\Pi$) are identified by method name alone. Similarly, fields ($\mathcal{F}$) are referenced via $lookup\text{-}field_\Pi$.

Semantics. The semantical domains of ADL programs are defined by a set of locations $\mathcal{L}$, numerical values $\mathcal{N}$ and a special value void, which together form the set of values $\mathcal{V}$. Objects carry their class name and a partial function from fields to values forming the set $\mathcal{O}$; similarly, arrays carry their length and a partial function from indices in $\mathbb{N}$ to values. Given a set of registers $\mathcal{X}$ (including two reserved registers from a set $\mathcal{X}_{res}$, see below), the register state is defined as $\mathcal{R} = (\mathcal{X}\cup\mathcal{X}_{res}) \to \mathcal{V}$. The space of heaps is

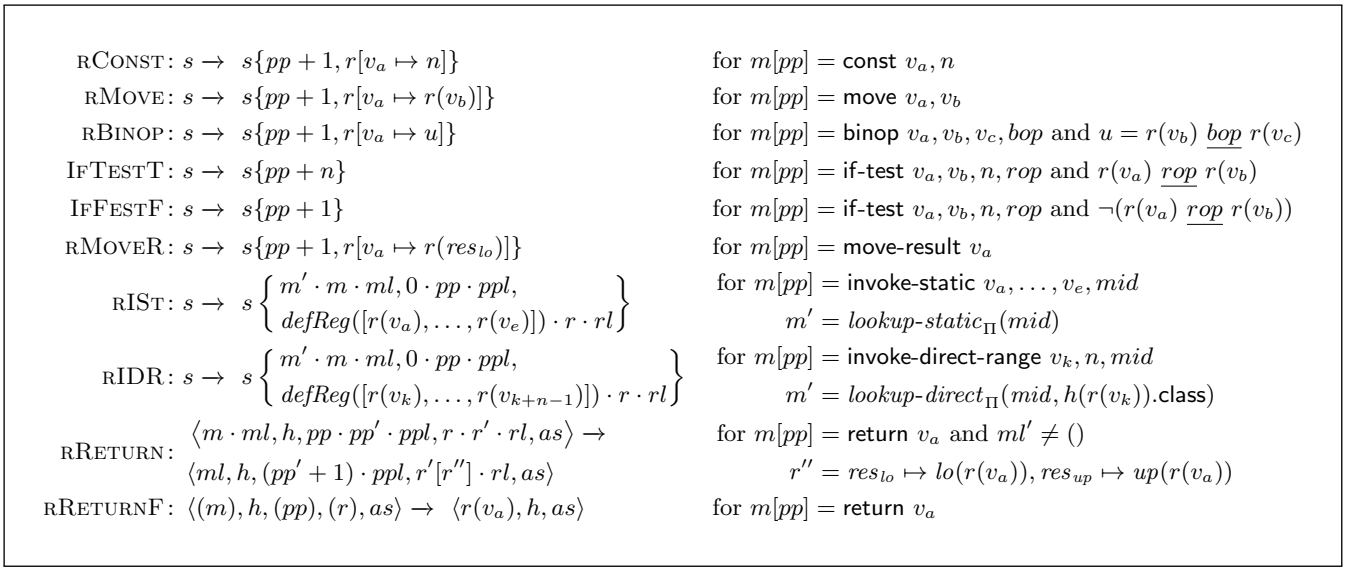


Figure 3: Selection of inference rules that define $\rightarrow$ with $uop \in \mathcal{UNOP}, bop \in \mathcal{BINOP}, rop \in \mathcal{RELOP}$, with some $n \in \mathbb{N}$ counter for method calls, some ADL program Π, and some method m. We write $v_a, \ldots, v_e$ for v_a, v_b, v_c, v_d, v_e, and if $r = \mathit{defReg}(u_1, \ldots, u_l)$, then $r(v_i) = u_i$ for $i \in \{0, \ldots, l\}$ and void otherwise.

$\mathcal{H} = (\mathcal{L} \rightharpoonup (\mathcal{O} \cup \mathcal{AR}))$ where $\mathcal{AR} = \mathbb{N} \times (\mathbb{N} \rightharpoonup \mathcal{V})$. A configuration describes the state of the program without the attacker: $\mathcal{C}' := \mathcal{M} \times \mathcal{H} \times \mathbb{N} \times \mathcal{R} \times \mathcal{Q}$. The full intermediate states additionally contains the attacker states $\mathcal{Q}$ (see Section 5): $\mathcal{C} = \mathcal{M} \times \mathcal{H} \times \mathbb{N} \times \mathcal{R} \times \mathcal{Q}$. Final states are defined as: $(\mathcal{C}_{\text{final}} = \mathcal{V} \times \mathcal{H} \times \mathcal{Q} \cup \mathcal{ADVR})$, where $\mathcal{ADVR}$ denotes the set of malicious functions (see Section 5).

Throughout the paper, we explicitly distinguish between configurations and states, in particular initial states and initial configurations.

The execution relation $\rightarrow$. The operational semantics is defined in terms of an execution relation $\rightarrow$ (for an ADL program Π, which we assume fixed in this section). For the sake of illustration, Figure 3 contains a representative selection of the rules defining $\rightarrow$. For the full set of rules, we refer to the work of Lortz et al. [4].

We use the following notation to shorten presentation and highlight the modifications applied to the state. For a state $s = \langle m \cdot ml, h, pp \cdot ppl, r \cdot rl, as \rangle \in \mathcal{C}$, we use $s\{pp+1\}$ to denote $\langle m \cdot ml, h, pp+1 \cdot ppl, r \cdot rl, as \rangle$. Similarly $s\{r[v \mapsto a]\}$ denotes $\langle m \cdot ml, h, pp \cdot ppl, r[v \mapsto a] \cdot rl, as \rangle$, $s\{h[l \mapsto a]\}$ denotes $\langle m \cdot ml, h[l \mapsto a], pp \cdot ppl, r \cdot rl, as \rangle$, $s\{m'\}$ denotes $\langle m' \cdot ml, h, pp \cdot ppl, r \cdot rl, as \rangle$, and $s\{as'\}$ denotes $\langle m \cdot ml, h, pp \cdot ppl, r \cdot rl, as' \rangle$.

The relation defines constant assignment (rConst), copying of register values (rMove), binary operations (rBinop), conditional branching (rIfTestTrue and rIfTestFalse). Moreover, we depict rules for static method evaluation (rISt) and evaluation of final methods (rIStR). Return values are stored in distinct result register $\mathit{res}_{lo}, \mathit{res}_{up} \in \mathcal{X}_{res}$ (see rReturn).

We slightly diverge from the characterization of method calls in [4] to capture the total number of computation steps in a run: each transition corresponds to one computation step, and each rule in Figure 3 is annotated accordingly ($\rightarrow$ instead of $\rightarrow_\Pi$).

5. SECURITY FRAMEWORK

We extend ADL with prbabilistic choices and with a probabilistic polynomial-time attacker that is invoked whenever specific functions are invoked. The modifications to the ADL-semantics are depicted in Figure 4.

Execution and communication model. ADL as defined in [4] does not support probabilistic choice and hence no generation of cryptographic keys and no executions of cryptographic functions. We thus first extended ADL with a rule that uniformly samples a register value from the set of numerical values $\mathcal{N}$ (see the Prob-rule in Figure 4).

To simplify presentation, we then interpret ADL's semantics as a probabilistic transition system in the following manner. A probabilistic transition system $T = (\mathrm{S}, s_0, A, \delta)$, defines probabilistic and non-deterministic transitions that are annotated with a number of computation steps, i.e., $\delta \colon S \rightarrow \mathcal{D}(S \times \mathbb{N}) \uplus \mathcal{P}(A \times S \times \mathbb{N})$. We write $s \xrightarrow{[p]}_n s'$ if $\delta(s) = \mu$ and $\mu(s', n) = p$, and $s \xrightarrow{a}_n s'$ if $(a, s', n) \in \delta(s)$. If $p = 1$ or $n = 1$, we omit $[p]$ or n, respectively. If S is finite or countably infinite, and T is *fully probabilistic*, i.e., there is no non-deterministic transition with more than one follow-up state, we introduce the following random variables: $\Pr[s \xrightarrow{\alpha}_n s'] = p$ iff. $s \xrightarrow{[p]}_n s' \wedge \alpha = \epsilon$ or $s \xrightarrow{\alpha}_n s' \wedge p = 1$. Given any initial state $s_0 \in \mathrm{S}^0$, we define the *outcome probability* of an execution $\Pr[\mathrm{Exec}(T) = s_0 \xrightarrow{\alpha_1}_{n_1} \cdots \xrightarrow{\alpha_n}_{n_n} s_n] = \prod_{i=0}^{n-1} \Pr[s_i \xrightarrow{\alpha_{i+1}} s_{i+1}]$ and the *probability of a trace* $(\alpha_1, \ldots, \alpha_n)$ as $\Pr[\mathit{Traces}(T) = (\alpha_1, \ldots, \alpha_n)] = \sum \Pr[\mathrm{Exec}(T) = s_0 \xrightarrow{\alpha_1}_{n_1} \cdots \xrightarrow{\alpha_n}_{n_n} s_n]$.

Threat model. In this work, we consider adversaries that are network parties or malicious apps that try to retrieve sensitive information from an honest app. We model the adversary as an external entity that cannot run any code within the program and that does not have access to the program's heap, but can read input to and control output

from a given set of *malicious functions* $\mathcal{ADVR}$. The attacker is a probabilistic polynomial-time algorithm $\mathcal{A}$.

We represent the attacker $\mathcal{A}$ in ADL as an unlabelled probabilistic transition system. We assume that each state is a triple, where the first element solely contains the inputs and the last element contains the outputs. We use the relation $\rightarrow_A$ to denote transitions in the attacker's transition system in Figure 4. Many computation models can be expressed this way, including, but not restricted to, Turing machines.

Whenever a malicious function is invoked, $\mathcal{A}$ is executed with its previous state *as* and the arguments *arg* of the malicious function. The output of $\mathcal{A}$ is the new state as' and a response-message *res* in $\mathcal{V}$, or in the set $\mathcal{ADVR}$, which is distinct from $\mathcal{V}$. If $res \in \mathcal{V}$, it is interpreted as the function's output values, otherwise, i.e., if the response message is in $\mathcal{ADVR}$, the execution terminates with the adversarial output $res \in \mathcal{ADVR}$. Figure 4 precisely defines this behavior in the rule rInvoke-Adv.

We can cast the ADL semantics in terms of a probabilistic transition system $\mathrm{ADL}_{\Pi,\mathcal{A},\langle ml,h,ppl,rl\rangle}$ that is parametric in the program executed, the attacker (see below) and the initial configuration. We obtain the following definition for executing ADL in the presence of an adversary.

Definition 3 (ADL Execution with Adv). *Given an ADL program* Π*, an adversary A, and an initial configuration* $\langle ml, h, ppl, rl\rangle$*, the probability that the interaction between* Π *on* $\langle ml, h, ppl, rl\rangle$ *and* $\mathcal{A}$ *terminates within n steps and results in x is defined as*

$$\Pr[\langle \Pi\, \langle ml, h, ppl, rl\rangle \,\|\mathcal{A}\rangle \downarrow_n x] := \Pr[\mathrm{ADL}_{\Pi,\mathcal{A},\langle ml,h,ppl,rl\rangle} \downarrow_n \langle x\rangle].$$

where we write (given $\mathcal{A}$*'s initial state as)* $\Pi\,\langle ml, h, ppl, rl\rangle$ *for the program* Π *with initial state* $\langle ml, h, ppl, rl, as\rangle$*.*

Calls to crypto APIs. Libraries that offer cryptographic operations leave the generation of randomness either implicit or explicit. Our result encompasses both settings. Moreover, most classes for cryptographic operations, such as the commonly used `javax.crypto.Cipher`, are used with a method call that lacks some arguments, e.g., the encryption key when using `javax.crypto.Cipher.doFinal()` to account for situations in which the missing argument has been stored earlier in a member variable, e.g., using `javax.crypto.Cipher.init()`. We address this by considering the class instance (from the heap) as an additional argument to the method call that has been marked, e.g., `javax.crypto.Cipher.doFinal()`. Since the class instance contains more information than the arguments of the cryptographic operation of the computationally sound symbolic model, all relevant information such as the encryption key, plaintext, and randomness has to be extracted from the class instance. The derived functions that we explicitly added in our formalization take care of this extraction.

Definition 4 (Library specification). *A library specification is an efficiently computable partial function* $libSpec : \mathcal{MID} \times \mathcal{O} \cup \mathcal{UNOP} \cup \mathcal{BINOP} \cup \mathcal{RELOP} \rightharpoonup \mathrm{SO}$ *defined at least on* $\mathcal{UNOP} \cup \mathcal{BINOP} \cup \mathcal{RELOP}$*.*

In order to be able to use an asymptotic security definition, we define uniform families of ADL programs as programs generated from a security parameter.

Definition 5 (Uniform families of programs). *Let* Π *be an algorithm that, given a security parameter* η*, outputs an ADL program. We denote the output of this program* Π^η *and call the set of outputs of this program a* uniform family of ADL programs.

Next, we define initial configurations for families (indexed by a security parameter) of transition systems as states that are valid initial configuration for all security parameters.

Definition 6 (Initial configuration). *Given a family* $(T_\eta)_\eta$ *of transition systems. We say that a state s is an* initial configuration *for* $(T_\eta)_\eta$ *if it is a valid initial configuration for all* T_η *in this family. Analogously, we say that a configuration s is an initial configuration for a uniform family of ADL programs if for all* η *this s is a valid initial configuration for* Π^η*.*

Indistinguishability of two ADL programs. Similar to CoSP, we define indistinguishability of ADL programs using tic-indistinguishability [13].

Definition 7 (Indistinguishability (ADL)). *We call two uniform families of ADL programs* $\Pi_1 = \{\Pi_1^\eta\}_{\eta\in\mathbb{N}}$ *and* $\Pi_2 = \{\Pi_2^\eta\}_{\eta\in\mathbb{N}}$ *with initial configuration* $s_1 = \langle ml_1, h_1, ppl_1, rl_1\rangle$ *and* $s_2 = \langle ml_2, h_2, ppl_2, rl_2\rangle$ computationally indistinguishable *for a (not necessarily uniform) family of attackers* $\mathcal{A} = \{\mathcal{A}^\eta\}_{\eta\in\mathbb{N}}$ *(written* $\Pi_1\langle s_1\rangle \approx_c^{\mathrm{ADL}^{\mathcal{A}}} \Pi_2\langle s_2\rangle$*) if for all polynomials p, there is a negligible function* μ *such that for all* $a, b \in \{0, 1\}$ *with* $a \neq b$*,*

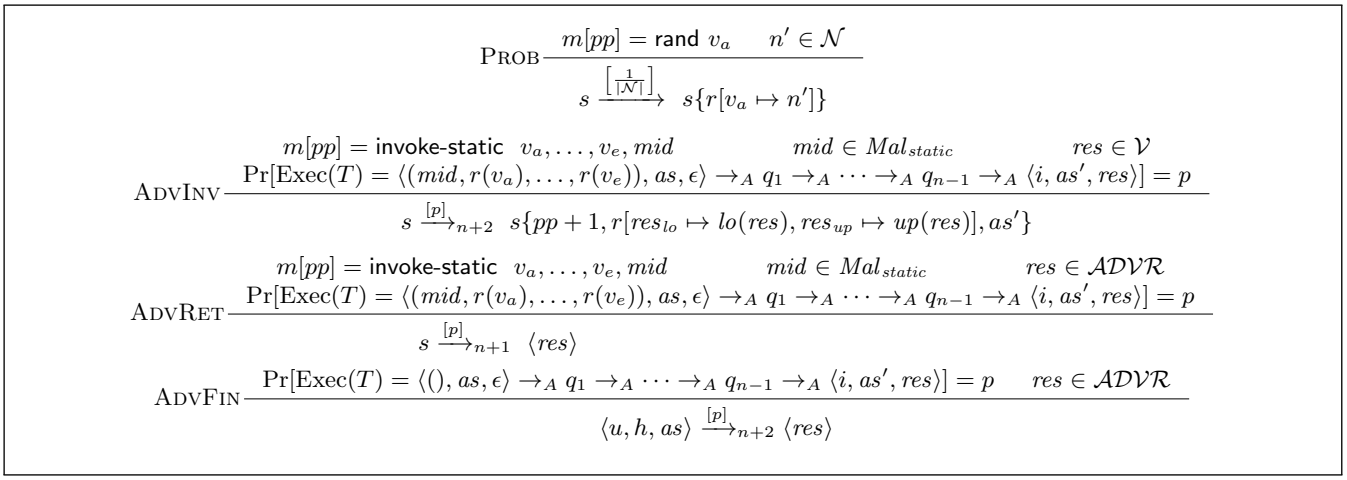
$$\Pr[\langle \Pi_1^\eta\langle s_1\rangle \|\mathcal{A}^\eta\rangle \downarrow_{p(\eta)} a] + \Pr[\langle \Pi_2^\eta\langle s_2\rangle \|\mathcal{A}^\eta\rangle \downarrow_{p(\eta)} b] \leq 1 + \mu(\eta).$$

We call $\Pi_1\langle s_1\rangle$ *and* $\Pi_2\langle s_2\rangle$ computationally indistinguishable $(\Pi_1\langle s_1\rangle \approx_c^{\mathrm{ADL}} \Pi_2\langle s_2\rangle)$ *if we have* $\Pi_1\langle s_1\rangle \approx_c^{\mathrm{ADL}^{\mathcal{A}}} \Pi_2\langle s_2\rangle$ *for all machines* $\mathcal{A}$*.*

This notion indistinguishability gives rise to notion of noninterference, when $\Pi_1 = \Pi_2$.

6. SYMBOLIC DALVIK BYTECODE

In this section, we define a symbolic variant of ADL, which uses symbolic terms instead of cryptographic values. We sometimes abbreviate this symbolic variant as ADLs. We will show in the next section that it suffices to analyze a (symbolic) ADLs program in order to prove the corresponding (cryptographic) ADL program secure, provided that the cryptographic operations used in that program are computationally sound, i.e., provided that they have a computationally sound symbolic model in CoSP. ADLs' semantics precisely corresponds to the semantics of ADL, except for the treatment of cryptographic operations. As a consequence, existing automated analysis tools can be conveniently extended to ADLs and, thus, accurately cope with cryptographic operations, since this extension only requires semantic adaptations precisely for those cases where cryptographic behavior needs to be captured. ADLs is parametric in the symbolic model of the considered cryptographic operations and, hence, benefits from the rich set of cryptographic primitives that are already supported by computational soundness results in CoSP, such as encryption and signatures [6–8, 10] or zero-knowledge proofs [9, 11].



$$\textsc{Prob}\ \frac{m[pp] = \mathsf{rand}\ v_a \qquad n' \in \mathcal{N}}{s \xrightarrow{\left[\frac{1}{|\mathcal{N}|}\right]} s\{r[v_a \mapsto n']\}}$$

$$\textsc{AdvInv}\ \frac{\begin{array}{c} m[pp] = \mathsf{invoke\text{-}static}\ \ v_a, \ldots, v_e, mid \qquad mid \in Mal_{static} \qquad res \in \mathcal{V} \\ \Pr[\mathrm{Exec}(T) = \langle (mid, r(v_a), \ldots, r(v_e)), as, \epsilon\rangle \to_A q_1 \to_A \cdots \to_A q_{n-1} \to_A \langle i, as', res\rangle] = p \end{array}}{s \xrightarrow{[p]}_{n+2} s\{pp+1, r[res_{lo} \mapsto lo(res), res_{up} \mapsto up(res)], as'\}}$$

$$\textsc{AdvRet}\ \frac{\begin{array}{c} m[pp] = \mathsf{invoke\text{-}static}\ \ v_a, \ldots, v_e, mid \qquad mid \in Mal_{static} \qquad res \in \mathcal{ADVR} \\ \Pr[\mathrm{Exec}(T) = \langle (mid, r(v_a), \ldots, r(v_e)), as, \epsilon\rangle \to_A q_1 \to_A \cdots \to_A q_{n-1} \to_A \langle i, as', res\rangle] = p \end{array}}{s \xrightarrow{[p]}_{n+1} \langle res\rangle}$$

$$\textsc{AdvFin}\ \frac{\Pr[\mathrm{Exec}(T) = \langle (), as, \epsilon\rangle \to_A q_1 \to_A \cdots \to_A q_{n-1} \to_A \langle i, as', res\rangle] = p \qquad res \in \mathcal{ADVR}}{\langle u, h, as\rangle \xrightarrow{[p]}_{n+2} \langle res\rangle}$$

Figure 4: Inference rules extending ADL with probabilistic semantics and adversarial interaction.

6.1 Embeddable symbolic CoSP models

We first define sufficient conditions under which a given CoSP model can be embedded into ADL. To this end, we require that, e.g., values in $\mathcal{V} = \mathcal{N} \cup \mathcal{L} \cup \{\text{void}\}$ can be embedded into CoSP, i.e., there needs to be an injective function from $\mathcal{V}$ into $\mathbf{T}$. Formally, we assume that there exists an injective function ι such that all $v \in \mathcal{V}$ can be distinguished using deconstructors, i.e., $\forall v, v' \in \mathcal{V} \exists D_1, \ldots, D_m \in \mathbf{D}$, $\iota(v') = v$ if $v' = v$ and $\perp$ otherwise. As in previous embeddings [6,14], we require that the symbolic model includes an equality destructor *equals* and a pairing constructor *pair*. We define $\widehat{v} = \iota(v)$ if $n \in \mathcal{V}$ and the identity if $n \in \mathbf{T}$, and lift this notion to sets: $\forall V \subseteq \mathcal{V}.\widehat{V} = \{\widehat{v} \mid v \in v\}$.

6.2 Semantics of symbolic ADL

The semantical domains of the symbolic variant of ADL coincide with the semantical domains of ADL except for the set of values $\mathcal{V}$ and the set of (intermediate and final) states. The set of values is extended to hold terms in addition to ADL values: $\mathcal{V} = \mathcal{N} \cup \mathcal{L} \cup \{\text{void}\} \cup \mathbf{T}$.

We add to each state in the inference rules the list of terms output so far, called *the symbolic view*. This symbolic view corresponds to the symbolic view in the symbolic execution of a CoSP protocol (Definition 1), and is used to track the information the adversary can use to deduce messages.

Transition between states are now annotated with the information sent to or received from the adversary. As ADL is sequential, every output to the adversary is followed by the adversary's input. Calls to malicious functions reveal register values to the adversary, which can be terms, locations, or numerical values. Terms are added to the adversarial knowledge as they are; locations and values are translated by means of the embedding. Calls to the crypto-API immediately yield a term by applying a constructor or destructor to the term arguments, or their symbolic representation. As elaborated in the introduction, applying binary operations (e.g., XOR) to symbolic terms usually invalidates computational soundness results. We over-approximate this by treating symbolic terms as a blackbox and let the outcome of any operation involving a symbolic term be decided by the adversary. As a side effect, we let the adversary learn the operands. As binary operations and tests on bits are only defined on inputs in $\mathcal{N}$, rules involving these remain unaltered and are complemented by the rules in Figure 5. For space reasons, the figure does not contain the rules for binary operations (Binop), but they are defined analogously to unary operations. For brevity, we let V' denote the attacker's view, which is part of the successor state, and write $V' \vdash m$, if the adversary can deduce a term m from a view V, i.e., there is a symbolic operation O such that $O(Out(V)) = m$. Observe that for IfTest-m, the adversary can decide the outcome without learning the operands. This highlights the non-determinism inherent to the symbolic semantics. Let $\mathrm{ADLs}_{\Pi,\langle ml,h,ppl,rl\rangle}$ denote the probabilistic transition system describing these modified semantical domains and transitions.

With the symbolic semantics in place, we can finally define the notion of symbolic equivalence between two programs. We first specify what constitutes a symbolic view, as our definition of traces applies only to fully probabilistic transition systems.

Definition 8 (Symbolic view). *Given a probabilistic (but not necessarily fully probabilistic) transition system T with initial state s_0, the set of symbolic views of T is defined:*

$$\mathrm{SViews}(T) = \left\{ (\alpha_1, \ldots, \alpha_m)|_{Views} \,\middle|\, s_0 \xrightarrow{\alpha_1}_{n_1} \cdots \xrightarrow{\alpha_m}_{n_m} s_m \right\}$$

We use the notion of symbolic equivalence introduced in Section 3.

Definition 9 (Symbolic equivalence). *Let $\mathbf{M}$ be a symbolic model. Two probabilistic transition systems T_1 and T_2 are* symbolically equivalent *($T_1 \approx_s^{\mathrm{SS}} T_2$) if* $\mathrm{SViews}(T_1) \sim \mathrm{SViews}(T_2)$ *w.r.t. $\mathbf{M}$. Two uniform families of ADL programs $\Pi_1 = \{\Pi_1^\eta\}_{\eta\in\mathbb{N}}$ and $\Pi_2 = \{\Pi_2^\eta\}_{\eta\in\mathbb{N}}$ and initial configurations $s_1 = \langle ml_1, h_1, ppl_1, rl_1\rangle$ and $s_2 = \langle ml_2, h_2, ppl_2, rl_2\rangle$ are* symbolically equivalent *($\Pi_1\langle s_1\rangle \approx_s^{\mathrm{ADL}} \Pi_2\langle s_2\rangle$) iff, for each η, the probabilistic transition systems $T_1 = \mathrm{ADLs}_{\Pi_1,s_1}$ and $T_2 = \mathrm{ADLs}_{\Pi_2,s_2}$ are symbolically equivalent.*

7. COMPUTATIONAL SOUNDNESS

Establishing a computational soundness proof for ADL requires a clear separation between honest program parts

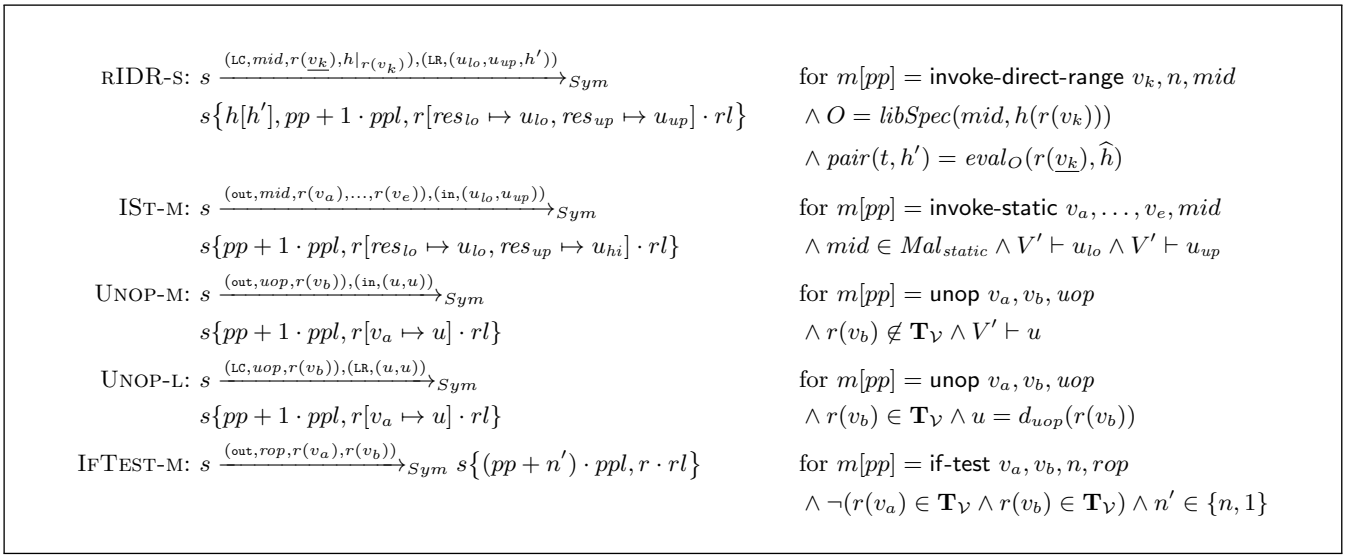


Figure 5: Modified inference rules of the symbolic execution relation $\rightarrow$ -Sym, where $s = \langle ml, h, pp\cdot ppl, r\cdot rl\rangle$.

and cryptographic API calls with their corresponding augmented adversarial symbolic capabilities. To achieve this, we characterize this partitioning by introducing the concept of a *split-state form* of an operational semantics, and we subsequently show that it can be naturally used to represent ADL. This section, presents the essence of our proof. For the full proofs and detailed definition, we refer to the technical report [12].

7.1 Split-state semantics

The split-state form partitions the original semantics into three components, parallely executed asynchronously: (*i*) all steps that belong to computing cryptographic operations (called the *crypto-API semantics*), (*ii*) all steps that belong to computing the malicious functions (called the *attacker semantics*), and (*iii*) all steps that belong to the rest of the program (called the *honest-program semantics*). The overall operational semantics is a composition of each of these sub-semantics, i.e., the state space is the Cartesian product of the sub-states with additional information on which entity is currently running, defined in terms of asynchronous parallel composition. We refer to the technical report for details [12]. In a split-state form, all three entities can synchronize through the following sets of labels: libCall, libResp, out, and in. The first two model message passing between the honest program and the crypto-API; the latter two model message passing between the honest program and the attacker. There is no synchronization step between attacker and the crypto-API, as the attacker can only stop the execution with a transition of the form $(\texttt{final}, m)$ for some message m.

An operational semantics is a *split-state semantics*, if it can be expressed as a transition system $(\mathrm{S}, \mathrm{S}^0, \rightarrow)$ by split-state composition of three transition systems.

The benefit of this notion is that it makes the communication explicit that occurs between the actual program and the adversary. This communication is often fixed but arbitrary, i.e., in the case of black-box usage. Moreover, it separates the actual program from the cryptographic API calls, which is crucial for computational soundness proofs. Finally, the details of the crypto-API are irrelevant for program analyses, provided that the cryptographic operations are implemented securely.

A split-state semantics is not necessarily a probabilistic transition systems, as the transition systems it is composed from might be non-deterministic. However, we define the probability of a certain outcome only for probabilistic split-state semantics, as in most applications for security, non-determinism in the honest program semantics stems from the modelling of concurrency and is usually resolved by specifying a scheduler. This non-determinism is typically conservatively resolved by assuming that the adversary controls the scheduling.

We write $\Pr[T(s_0) \downarrow_n x]$ for the probability that the interaction between honest program, attacker and crypto-API with initial state $s_0 \in \mathrm{S}^0$ results in x and terminates within n steps. Split-state indistinguishability is then defined as follows.

Definition 10 (Split-state indistinguishability). *Let $T_{H,1}$, $T_{H,2}$, T_A, and T_L be families of transition systems indexed by a security parameter in $\mathbb{N}$. We call $T_{H,1}$ and $T_{H,2}$* computationally indistinguishable *for T_L and T_A and initial states $s^0_{H,1}$, $s^0_{H,2}$, s_L of $T_{H,1}$, $T_{H,2}$, T_L (respectively) in the sense of Definition 6 if for all polynomials p, there is a negligible function μ such that for all $a, b \in \{0,1\}$ with $a \neq b$,*

$$\Pr[T_1^\eta((\top, s^0_{H,1}), (\bot, (0,z)), (\bot, s_L)) \downarrow_{p(\eta)} a] + \Pr[T_2^\eta((\top, s^0_{H,2}), (\bot, (0,z)), (\bot, s_L)) \downarrow_{p(\eta)} b] \leq 1 + \mu(\eta),$$

where T_1^η is the split-state composition of $T^\eta_{H,1}, T^\eta_L, T_A$, and T_2^η is the split-state composition of $T^\eta_{H,2}, T^\eta_L, T_A$ for security parameter η. In short, we write $T_{H,1}(s^0_{H,1}) \approx_c^{T_L(s_L), T_A} T_{H,2}(s^0_{H,2})$. Additionally, we call $T_{H,1}$ and $T_{H,2}$ computationally indistinguishable (written as $T_{H,1}(s^0_{H,1}) \approx_c^{T_L(s_L)} T_{H,2}(s^0_{H,2})$) *if $T_{H,1}(s^0_{H,1}) \approx_c^{T_L(s_L), T_A} T_{H,2}(s^0_{H,2})$ for all $T_A \in \mathcal{ADV}'$ and for initial states $s^0_{H,1}, s^0_{H,2}, s_L$ for $T_{H,1}, T_{H,2}, T_L$ (respectively).*

7.2 Split-state representation of ADL

We now define the condition necessary for splitting an ADL program into an honest program semantic and a crypto-API semantics. For a formal definition, we refer to the technical report [12]. An ADL program Π is *pre-compliant* with a crypto-API specification if: (i) the crypto-API only reads the slice of the heap belonging to its object, i.e., only objects reachable from locations stored in its own member variables, (ii) the crypto-API never invokes the adversary, and (iii) only the crypto-API and the adversary make use of the rand-instruction.

DEFINITION 11 (ADL SPLIT-STATE REPRESENTATION). *Given a uniform family of ADL programs $\Pi = \{\Pi^\eta\}_{\eta\in\mathbb{N}}$, every member thereof pre-compliant with a crypto-API specification libSpec, a family of attackers $\{\mathcal{A}^\eta\}_{\eta\in\mathbb{N}} \in \mathcal{ADV}$, and an initial configuration $s_0 = \langle ml, h, ppl, rl\rangle$, we define the ADL split-state representation $(\mathrm{ADL}^{SS}_{\Pi^\eta,\mathcal{A}^\eta,s_0})_{\eta\in\mathbb{N}}$ as the family of split-state compositions of the following three transition systems for every $\eta \in \mathbb{N}$:*

- honest-program semantics: *The transition system $(\mathrm{S}_H, s^0_H, A_H, \delta_H)$ is defined by the ADL semantics, extended with rules in Figure 6, and the initial state $\langle ml, h, ppl, rl, as^H_{dmy}\rangle$ for an arbitrary initial adversary state as^H_{dmy} which will be ignored.*
- crypto-API semantics: *The transition system $(\mathrm{S}_L, s^0_L, A_L, \delta_L)$ is defined by the ADL semantics except* rReturnVF, rReturnF, *extended with rules in Figure 4 and* Prob *(see Figure 7). The initial state is $\langle (), \emptyset, (), (), as^L_{dmy}\rangle$ for some arbitrary initial adversary state as^L_{dmy} which will be ignored.*
- attacker semantics: *The transition system $(\mathrm{S}_A, s^0_A, A_A, \delta_A)$ consists of the transitions of the adversary $\mathcal{A}$, extended with the transitions in Figure 8.*

To make our notation more concise, we abbreviate the initial state $(\top, \langle ml, h, ppl, rl, as^H_{dmy}\rangle), (\bot, s_A), (\bot, \langle (), \emptyset, (), (), as^L_{dmy}\rangle)$ for some honest initial state $\langle ml, h, ppl, rl\rangle$ as $\langle ml, h, ppl, rl\rangle^{ss}$.

LEMMA 1. *For all ADL programs Π that are pre-compliant with a crypto-API specification libSpec, all initial configurations $\langle ml, h, ppl, rl\rangle$, all adversaries all $n \in \mathbb{N}$, we have*

$$\Pr[\langle \Pi\, \langle ml, h, ppl, rl\rangle \,\|\mathcal{A}\rangle \downarrow_n x] = \Pr[T \downarrow_n x],$$

where T denotes the ADL split-state representation of Π and $\mathcal{A}$ for initial configuration $\langle ml, h, ppl, rl\rangle$.

7.3 Over-approximating ADL

Recall that our plan is to instantiate $\mathcal{D}_c$ with symbolic terms and even with positions in a CoSP tree. However, some operations are not defined on symbolic terms. As an example consider the XOR operation. It has been shown that each computationally sound symbolic representation of XOR has to work on symbolic bitstrings. Hence, for a typical symbolic representation of a ciphertext the XOR operation is not defined in the symbolic model.

In order to get rid of any undefined operations after instantiating $\mathcal{D}_c$ with symbolic terms or CoSP tree positions, we over-approximate each undefined operation by querying the attacker for the result. To this end, we define an over-approximation of the ADL semantics in terms of its split-state representation. This over-approximation tracks values resulting from calls to the crypto-API. Whenever a computation, e.g., a unary operation, a binary operation or a test, is performed on these values, the over-approximation gives the adversary more power: she can decide the values of these computations. This is necessary, as these operations cannot be computed in the symbolic model. Already performing this over-approximation on the ADL split-state semantics simplifies the embedding. Thanks to the split-state composition, we can define this over-approximation canonically. Yet for space constraints, we refer the reader to the technical report [12], and give here only the core of it.

We assume some domain $\mathcal{D}$ underlying the transition function and the set of states of the honest program semantic, and a distinct new domain $\mathcal{D}_c$ (for values resulting from the crypto API), a subset of which $\mathcal{D}_b \subset \mathcal{D}_c$ (representation of bitstrings) have a bijection to the original domain (see Figure 9). We assume the honest-program semantics to be defined via a set of rules, and interpret these rules in the new domain $\mathcal{D}_c$. If the re-interpreted rule can be instantiated regardless of whether some state carries values in $\mathcal{D}$ or $\mathcal{D}_c$, this rule is transferred to the over-approximated semantics. Most rules only move values from registers to heaps and are homomorphic in this sense. Rules for which the above do not hold, but which can be expressed using a constructor or destructor, are split into four rules, two of which send these values to the adversary and use her input for the follow-up state in case one of the variables is in $\mathcal{D}_c\backslash\mathcal{D}_b$. Otherwise, i.e., if only values from $\mathcal{D}$ are used, or values representable in $\mathcal{D}$, then the crypto-API is used for the computation. This way, we were able to encode binary operations like XOR as destructors, so representations of bitstrings that have passed the crypto API (e.g., a bitstring was encrypted and then decrypted again) can be treated without unnecessary imprecision. If any rule $r \in R_H$ falls in neither of the above cases, the canonical over-approximation is undefined.

Using this over-approximation, we can derive ADLo^{SS}, the over-approximated ADL semantics. We tag messages resulting from calls to the crypto-API using a set $\mathcal{N}_c$, such that $\mathcal{N}_c \cap \mathcal{N} = \emptyset$, and assume a bijection between the two that is efficiently computable. For $n \in \mathcal{N}_c \cup \mathcal{N}$, let $[n]_{\mathcal{N}_c}$ and $[n]_{\mathcal{N}}$ denote its representation in $\mathcal{N}_c$ or $\mathcal{N}$ according to the bijection. We lift this notation to values in $\mathcal{V}$, too. The inference rules include those previously defined, but modified such that register values are converted to $\mathcal{N}$ before addressing the crypto-API, i.e., $\mathtt{LC}(f, r(v_a), \ldots, r(v_e))$ is substituted by $\mathtt{LC}(f, [r(v_a)]_{\mathcal{N}_c}, \ldots, [r(v_e)]_{\mathcal{N}_c})$, and $\mathtt{out}(mid, r(v_a), \ldots, r(v_e))$ is substituted by $\mathtt{out}(mid, [r(v_a)]_{\mathcal{N}_c}, \ldots, [r(v_e)]_{\mathcal{N}_c})$. Finally, the rules in Figure 10 are added.

DEFINITION 12 (ADL SPLIT-STATE EQUIVALENCE $\approx^{\mathrm{SS}}_c$). *Let Π_1 and Π_2 be two families of ADL programs compliant with the same library specification libSpec and s_1, s_2 initial configurations for Π_1 and Π_2, respectively. We write $\Pi_1\langle s_1\rangle \approx^{\mathrm{SS}}_c \Pi_2\langle s_2\rangle$, if, for all adversaries $\mathcal{A} \in \mathcal{ADV}$, for the over-approximated ADL split-state representations $(\mathrm{ADLo}^{SS}_{\Pi^\eta_1,\mathcal{A}^\eta,s_1})_{\eta\in\mathbb{N}}$ of $\Pi_1\langle s_1\rangle$ and $\mathcal{A}$, and $(\mathrm{ADLo}^{SS}_{\Pi^\eta_2,\mathcal{A}^\eta,s_2})_{\eta\in\mathbb{N}}$ of $\Pi_2\langle s_2\rangle$ and $\mathcal{A}$, we have $T_{H,1}(s_1) \approx^{T_L,T_A}_c T_{H,2}(s_2)$.*

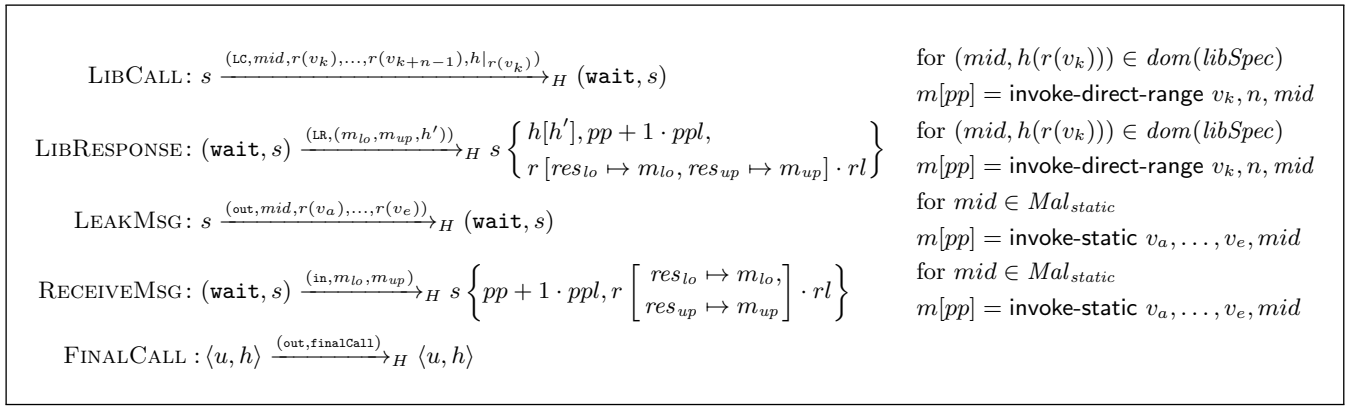


Figure 6: ADL split-state representation, honest program semantics, $s = \langle ml, h, ppl, r \cdot rl \rangle$

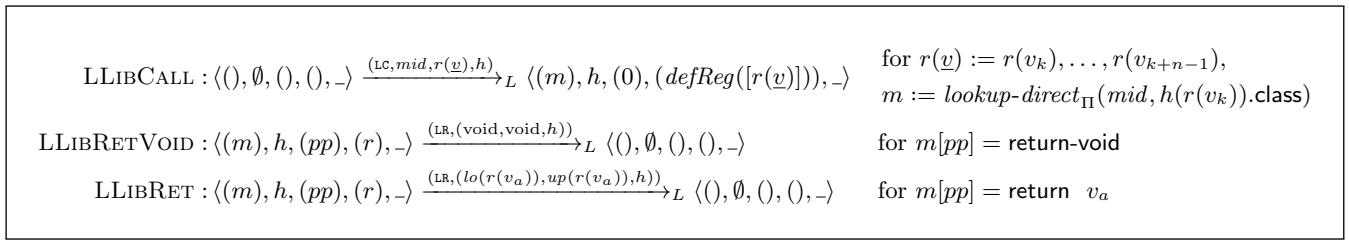


Figure 7: ADL split-state representation, library program semantics, _ is an arbitrary adversarial state.

Next, we state the lemma that connects the ADL semantics to the over-approximated split-state form.

Lemma 2. *Let Π_1 and Π_2 be two families of ADL programs pre-compliant with the same library specification libSpec and s_1, s_2 initial configuration for Π_1 and Π_2, respectively. Then $\Pi_1\langle s_1\rangle \approx_c^{\text{SS}} \Pi_2\langle s_2\rangle \implies \Pi_1\langle s_1\rangle \approx_c^{\text{ADL}} \Pi_2\langle s_2\rangle$.*

7.4 Canonical symbolic semantics

The canonical over-approximated split-state semantics directly defines a canonical symbolic semantics. We replace the domain $\mathcal{D}_c$ by the set of terms that are defined by the CoSP-symbolic model $\mathbf{M}$, we replace the attacker semantics by the symbolic attacker from $\mathbf{M}$, and we replace the crypto-API by the constructors and destructors in $\mathbf{M}$.

Definition 13 (Symbolic equivalence $\approx_s^{\text{SS}}$). *Two ADL programs Π_1 and Π_2, and initial configurations $s_1 = \langle ml_1, h_1, ppl_1, rl_1\rangle$ and $s_2 = \langle ml_2, h_2, ppl_2, rl_2\rangle$ are* symbolically split-state equivalent ($\Pi_1\langle s_1\rangle \approx_s^{\text{SS}} \Pi_2\langle s_2\rangle$) *if for all attacker strategies I, their respective ADL symbolic split-state semantics T_1, T_2 w.r.t. to I are symbolically equivalent, i.e., if* $\text{SViews}(T_1) \sim \text{SViews}(T_2)$.

Next, we state the lemma that connects symbolic ADL to the symbolic split-state composition of ADL.

Lemma 3. *Let Π_1 and Π_2 be two ADL program pre-compliant to the same library specification libSpec and s_1, s_2 initial configuration for Π_1 and Π_2, respectively. Then,*

$$\Pi_1\langle s_1\rangle \approx_s^{\text{ADL}} \Pi_2\langle s_2\rangle \implies \Pi_1\langle s_1\rangle \approx_s^{\text{SS}} \Pi_2\langle s_2\rangle$$

7.5 The CoSP-embedding

We are finally in the position to construct the embedding e from the part of the program that is encoded in the honest-program semantics into CoSP. Recall that the symbolic variant of ADL is also solely defined on the honest-program semantics of a program. Recall that we map the honest-program semantics T_H that belongs to an ADL program Π into CoSP. In the context of a pair of ADL programs Π_1, Π_2, we write $T_{H,i}$ for the honest program semantics for Π_i.

The main lemmas of this section show that for every pair of ADL programs Π_1, Π_2 with initial configurations s_1, s_2 the equivalence of $\Pi_1\langle s_1\rangle$ and $\Pi_2\langle s_2\rangle$ in symbolic ADL implies the symbolic equivalence of $e(T_{H,1}(s_1))$ and $e(T_{H,2}(s_2))$ in CoSP and computational indistinguishability of $e(T_{H,1})$ and $e(T_{H,2})$ in CoSP implies computational indistinguishability of $\Pi_1\langle s_1\rangle$ and $\Pi_2\langle s_2\rangle$.

First we construct the embedding, and then we give the main arguments why the embedding preserves symbolic equivalence and computational indistinguishability.

Constructing the embedding into CoSP. As CoSP trees are infinite, we define the embedding in a co-recursive manner, i.e., as the largest fixpoint of a co-recursive construction. Each step in this recursion is defined by a function $E_{T_H(s)}$ that takes as input a trace from a leaf-node in the so-far constructed CoSP tree to the root node and outputs a finite subtree. Within the embedding, we re-interpret the transition system of the honest program introduced in Section 7.3 by instantiating the set $\mathcal{D}_c$ to be the set of positions in a CoSP tree. With this representation, registers and heap locations can store values input by the adversary or the crypto-API by pointing to the position of the respective input or computation nodes in addition to numerical values, locations, and void. Whenever this kind of special register values are supposed to be given to the adversary or the crypto-API, the position is resolved to a node identifier.

$$\textsc{ALeakMsg} : \langle \varepsilon, as, \varepsilon \rangle \xrightarrow{(\texttt{out}, mid, v_a, \ldots, v_e)}_A \langle (mid, v_a, \ldots, v_e), as, \varepsilon \rangle$$

$$\textsc{AReceiveMsg} : \langle i, as, res \rangle \xrightarrow{(\texttt{in}, (lo(res), up(res)))}_A \langle \epsilon, as', \epsilon \rangle \qquad \text{for } res \in \mathcal{V}$$

$$\textsc{AFinal} : \langle i, as, res \rangle \xrightarrow{(\texttt{final}, res)}_A \langle \epsilon, as', \epsilon \rangle \qquad \text{for } res \in \mathcal{ADVR}$$

Figure 8: ADL split-state representation, adversary semantics, ε denotes the empty string.

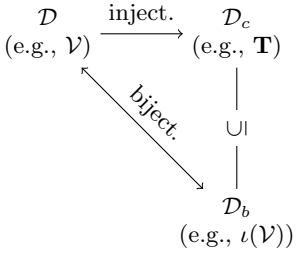


Figure 9: Domains of the canonical over-approx.

The recursion step $E_{T_H(s)}$ in the construction of the embedding internally runs the honest semantics of this modified version of the over-approximated split-state semantics. It computes the next state from deterministic (internal) transitions until it arrives at a non-deterministic (calling) transition of the honest-program semantics, i.e., either a `LC`- or a `out`-transition. Each call to the library (i.e., an `LC`-transition) is transformed into a computation node with two successors (labelled *yes* and *no*), referencing freshly constructed values or previous nodes depending on the information in the label of the transition. Each call to the adversary (i.e., an `out`-transition) is transformed into an output node, again referencing values or nodes based on the label of the transition, and an input node for the `in`-transition following suit.

Soundness of the embedding. The main idea in the proof is to consider the sub-sequences of the traces that start and end with calls to, or responses from the attacker or the library. Each step between the sub-sequences corresponds to an edge in the CoSP-tree. If we choose a canonical over-approximation such that the domain of the variable values is the set of positions in the CoSP-tree, then we can show the following relation: for the over-approximated split-state semantics, there is a direct correspondence between the states at the end of such a sub-sequence and the nodes in the CoSP-tree, which store the current state in the node-identifier.

In order to match the notation in the overview, we write $e(\Pi_i\langle s_i\rangle)$ for $e(T_{H,i}(s_i))$ in the following lemmas, for two ADL programs Π_1, Π_2 and two initial configurations s_1, s_2.

Lemma 4. *Let Π_1 and Π_2 two ADL program together with input configurations s_1, s_2 in the canonical symbolic model with respect to a CoSP-symbolic model $\mathbf{M}$. Let Π_1 and Π_2 be pre-compliant with the same library specification. Then,*

$$\Pi_1\langle s_1\rangle \approx_s^{\text{SS}} \Pi_2\langle s_2\rangle \implies e(\Pi_1\langle s_1\rangle) \approx_s^{\text{CoSP}} e(\Pi_2\langle s_2\rangle)$$

For our the next lemma and our main theorem, we require that two ADL programs are *compliant*, i.e., the programs are pre-compliant with the same library specification (see Section 7.2), and the library produces the same distribution as the implementation Impl from the computational soundness result with a library specification *libSpec*.

Lemma 5. *Let Π_1 and Π_2 two ADL program together with input configurations s_1, s_2 in the canonical symbolic model with respect to a CoSP-symbolic model $\mathbf{M}$. Let Π_1 and Π_2 be compliant with the same library specification w.r.t a symbolic model $\mathbf{M}$ and an implementation* Impl. *Then,*

$$e(\Pi_1\langle s_1\rangle) \approx_c^{\text{CoSP}} e(\Pi_2\langle s_2\rangle) \implies \Pi_1\langle s_1\rangle \approx_c^{\text{SS}} \Pi_2\langle s_2\rangle.$$

Our computational soundness result is parametric in a given symbolic model and given conditions C_I to the implementation and the protocols such that computational soundness in the sense of the CoSP framework holds. Our result states that for any symbolic model with conditions C in CoSP, equivalence in symbolic ADL (see Section 6) implies indistinguishability in ADL (see Section 5). Since all CoSP results in the literature characterize the protocol class by a set of protocol conditions C_P, we use these protocol conditions in our theorem as well.

Theorem 1. *Let a symbolic model $\mathbf{M}$, protocol conditions C_P and implementation conditions C_I that are computationally sound in the sense of CoSP (Definition 2) be given. Let Π_1 and Π_2 be two uniform families of ADL programs compliant with initial configurations s_1, s_2 and the same library specification libSpec w.r.t. to a symbolic model $\mathbf{M}$ and all implementations* Impl *that satisfy C_I. Then, the following implication holds*

$$\Pi_1\langle s_1\rangle \approx_s^{\text{ADL}} \Pi_2\langle s_2\rangle \implies \Pi_1\langle s_1\rangle \approx_c^{\text{ADL}} \Pi_2\langle s_2\rangle.$$

8. RELATED WORK

Operational semantics for Dalvik Bytecode. We have opted to ground our work on the Abstract Dalvik Language (ADL) [4]. ADL currently excels over alternative semantics such as the ones proposed by Wognsen et al. [15], Xia et al. [16], TaintDroid [3], and Chaudhuri [17] because of its comprehensive treatment of the Dalvik language, even though ADL currently only provides sequential executions and does not support exceptions. TaintDroid is more general in this respect in that it contains an ad-hoc modelling of concurrent execution. However, in the ADL extension put forward with the adversary that includes probabilistic choices and adversary interactions, concurrency in ADL can be modelled via program transformations, as we discuss in Section 7.

Information-flow control with cryptographic primitives. The standard notion of security in information flow control – non-interference – is too strong when cryptographic operations are being considered, as, e.g., an encryption of

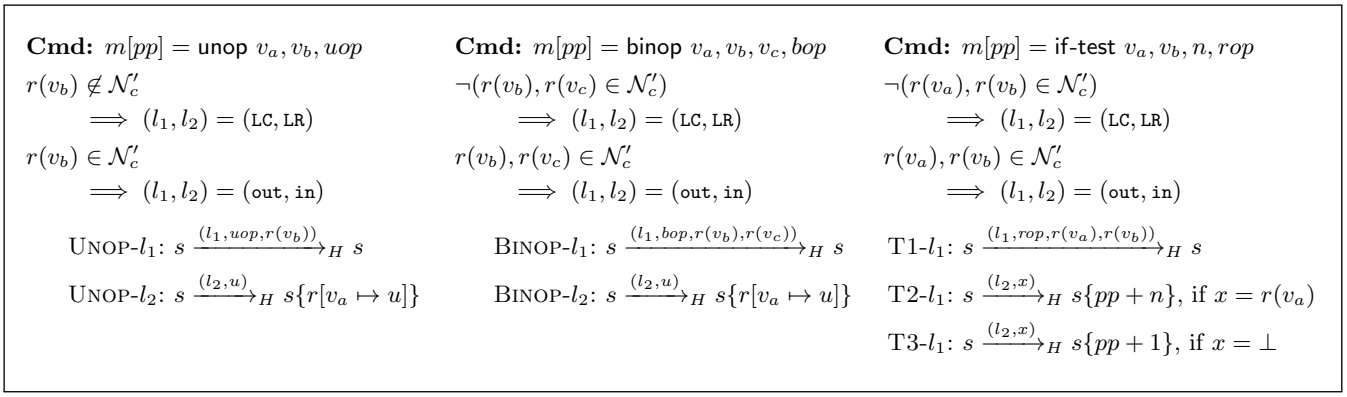


Figure 10: over-approximated honest program semantics for ADL (ADLo^{SS}). Here $\mathcal{N}_b := range(\iota)$ and $\mathcal{N}'_c := \mathcal{N}_c \setminus \mathcal{N}_b$.

a secret key and a secret message is intuitively safe to be stored in a public variable, but it nontheless results in different values depending on the key and the message. While the declassification of values (see [18] for an introduction and overview of results) can be used to relax this notion, it is difficult to decide under what circumstances cryptographic values can be safely declassified.

Our approach is similar to Askarov et al.'s, permitting so-called *cryptographically masked flows* by considering a relaxed equivalence notion on public values, masking acceptable information flow when ciphertexts are made public [19].

Laud has shown the computational soundness of this approach [20] if the employed encryption scheme satisfies key-dependent message security, provides plaintext integrity, and keys are only used in the correct key position, etc. As cryptographically masked flows are captured in a possibilistic setting, leaks through probabilistic behavior are not captured; hence the program must not be able to branch on probabilistic values. In our work, we treat computations performed on cryptographic values conservatively by relaying them to the adversary. Furthermore, previous work introduced security type-systems, such as [21], and program analyses, such as [22], that directly operate on the computational semantics and are thus capable of verifying noninterference in a probabilistic setting. This approach avoids the aforementioned problem at the cost of less modularity and less potential for automation than our approach.

Computational Soundness. On computational soundness, there is a rich body of literature for various cryptographic primitives [11, 23–25], with malicious keys [8, 26], and even composable computational results [27, 28]. Even though these works covered the applied π-calculus [6,26,29], the stateful applied π calculus [30] and RCF [14] and even a embedding of a fragment of C [31], none of these works provide a computational soundness result is known for Dalvik bytecode and hence for Android app analysis.

Another line of work does not provide a complete symbolic characterization of the attacker but concentrates on single rules that hold for certain symbolic terms [32, 33]. This restriction enables a much more flexible and composable preservation notion but pays with omitting any guarantee that the set of rules characterizes all attacker-actions. Hence, it is not clear how well-suited this approach is to automation.

Interactive proof assistants for cryptography. Cryptographic proof assistants enable the mechanized verification of cryptographic proofs, without first abstracting the cryptographic operations [34–36]. Consequently, these tools only offer limited automation. Yet complementarily, these tools could be used to verify that a library satisfies the conditions that a computational soundness result requires.

9. CONCLUSION AND FUTURE WORK

We have shown how cryptographic operations can be faithfully included into existing approaches for automated app analysis. Our results overcome the overly pessimistic results that arise when current automated approaches deal with apps that contain cryptographic operations, as these results do not account for secrecy properties offered by cryptographic operations such as encryption. We have shown how cryptographic operations can be expressed as symbolic abstractions within Dalvik bytecode, so that these abstractions can be conveniently added to existing app analysis tools using minor changes in their semantics. Moreover, we have established the first computational soundness result for the Abstract Dalvik Language (ADL) [4], which currently constitutes the most detailed and comprehensive operational semantics for Dalvik in the literature.

A result that we only scratched on in this work is that any small-step semantics expressed in our novel split-state form entails a canonical small-step semantics for a symbolic model that is computationally sound. This hence provides a recipe for establishing computationally sound symbolic abstractions for *any* given programming language, provided that one can show that the interaction with the attacker and the cryptographic operations can be expressed by means of our concept of split-state semantics. We plan to further investigate this claim and its applicability to modern programming languages in future work.

10. ACKNOWLEDGEMENTS

This work has been partially funded by the German Research Foundation (DFG) via the collaborative research center "Methods and Tools for Understanding and Controlling Privacy" (SFB 1223), project B3. This work has been partially supported by the Zurich Information Security Center (ZISC).

11. REFERENCES

[1] Symantec, "Internet security threat report, volume 20," Accessed: Oct 15, 2015. [Online]. Available: http://www.symantec.com/security_response/publications/threatreport.jsp

[2] GData, "Mobile malware report: Q2/2015," Accessed: Oct 15, 2015. [Online]. Available: https://public.gdatasoftware.com/Presse/Publikationen/Malware_Reports/G_DATA_MobileMWR_Q2_2015_EN.pdf

[3] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones," *Communications of the ACM*, vol. 57, no. 3, pp. 99–106, 2014.

[4] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber, "Cassandra: towards a certifying app store for android," in *Proc. 4th ACM SPSM*, 2014, pp. 93–104.

[5] M. Backes, S. Bugiel, E. Derr, S. Gerling, and C. Hammer, "R-droid: In-depth application vetting for android with path-sensitive value analysis," in *Proc. ACM AsiaCCS*, 2016.

[6] M. Backes, D. Hofheinz, and D. Unruh, "CoSP: A General Framework for Computational Soundness Proofs," in *Proc. 16th ACM CCS*, 2009, pp. 66–78.

[7] S. Meiser, "Computational soundness of passively secure encryption in presence of active adversaries," Master's thesis at Saarland University, 2010.

[8] M. Backes, A. Malik, and D. Unruh, "Computational Soundness without Protocol Restrictions," in *Proc. 19th ACM CCS*, 2012, pp. 699–711.

[9] M. Backes, F. Bendun, and D. Unruh, "Computational Soundness of Symbolic Zero-knowledge Proofs: Weaker Assumptions and Mechanized Verification," in *Proc. 2nd POST*, 2013, pp. 206–225.

[10] M. Backes, E. Mohammadi, and T. Ruffing, "Computational Soundness Results for ProVerif," in *Proc. 3rd POST*, 2014, pp. 42–62.

[11] M. Backes, F. Bendun, M. Maffei, E. Mohammadi, and K. Pecina, "A Computationally Sound, Symbolic Abstraction for Malleable Zero-knowledge Proofs," in *Proc. 28th IEEE CSF*, 2015, pp. 412–480.

[12] M. Backes, R. Künnemann, and E. Mohammadi, "Technical report: Computational soundness for dalvik bytecode," arXiv:1608.04362, 2016.

[13] D. Unruh, "Termination-Insensitive Computational Indistinguishability (and Applications to Computational Soundness)," in *Proc. 24th IEEE CSF*, 2011, pp. 251–265.

[14] M. Backes, M. Maffei, and D. Unruh, "Computationally Sound Verification of Source Code," in *Proc. 17th ACM CCS*, 2010, pp. 387–398.

[15] E. R. Wognsen, H. S. Karlsen, M. C. Olesen, and R. R. Hansen, "Formalisation and analysis of dalvik bytecode," *Science of Computer Programming*, vol. 92, pp. 25–55, 2014.

[16] M. Xia, L. Gong, Y. Lyu, Z. Qi, and X. Liu, "Effective real-time android application auditing," in *Proc. 36th IEEE S&*, 2015, pp. 899–914.

[17] A. Chaudhuri, "Language-based security on android," in *Proc. 4th ACM PLAS*, 2009, pp. 1–7.

[18] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *J. Computer Security*, vol. 17, no. 5, 2009.

[19] A. Askarov, D. Hedin, and A. Sabelfeld, "Cryptographically-masked flows," *TCS*, vol. 402, no. 2-3, pp. 82–101, 2008.

[20] P. Laud, "On the computational soundness of cryptographically masked flows," in *Proc. of 35th POPL*, 2008, pp. 337–348.

[21] P. Laud and V. Vene, "A type system for computationally secure information flow," in *Proc. of 15th FCT*, 2005, pp. 365–377.

[22] P. Laud, "Handling encryption in an analysis for secure information flow," in *Proc. of 12th ESOP*, 2003, pp. 159–173.

[23] M. Backes, B. Pfitzmann, and M. Waidner, "A Composable Cryptographic Library with Nested Operations," in *Proc. 10th ACM CCS*, 2003, pp. 220–230.

[24] V. Cortier and B. Warinschi, "Computationally Sound, Automated Proofs for Security Protocols," in *Proc. 14th ESOP*, 2005, pp. 157–171.

[25] D. Galindo, F. D. Garcia, and P. van Rossum, "Computational Soundness of Non-Malleable Commitments," in *Proc. 4th ISPEC*, 2008, pp. 361–376.

[26] H. Comon-Lundh, V. Cortier, and G. Scerri, "Security Proof with Dishonest Keys," in *Proc. 1nd POST*. Springer, 2012, pp. 149–168.

[27] V. Cortier and B. Warinschi, "A composable computational soundness notion," in *Proc. 18th ACM CCS*, 2011, pp. 63–74.

[28] F. Böhl, V. Cortier, and B. Warinschi, "Deduction Soundness: Prove One, Get Five for Free," in *Proc. 20th ACM CCS*, 2013, pp. 1261–1272.

[29] H. Comon-Lundh and V. Cortier, "Computational Soundness of Observational Equivalence," in *Proc. 15th ACM CCS*. ACM Press, 2008, pp. 109–118.

[30] J. Shao, Y. Qin, and D. Feng, "Computational Soundness Results for Stateful Applied Pi Calculus," in *Proc. 5rd POS*, 2016, pp. 254–275.

[31] M. Aizatulin, A. D. Gordon, and J. Jürjens, "Computational Verification of C Protocol Implementations by Symbolic Execution," in *Proc. 19th ACM CCS*, 2012, pp. 712–723.

[32] G. Bana and H. Comon-Lundh, "Towards Unconditional Soundness: Computationally Complete Symbolic Attacker," in *Proc. 1nd POST*, 2012, pp. 189–208.

[33] ——, "A Computationally Complete Symbolic Attacker for Equivalence Properties," in *Proc. 21th ACM CCS*, 2014, pp. 609–620.

[34] G. Barthe, B. Grégoire, S. Heraud, and S.-Z. Béguelin, "Computer-Aided Security Proofs for the Working Cryptographer," in *Proc. CRYPTO*, 2011, pp. 71–90.

[35] A. Petcher and G. Morrisett, "The Foundational Cryptography Framework," in *Proc. 4th POST*, 2015, pp. 53–72.

[36] A. Lochbihler, "Probabilistic functions and cryptographic oracles in higher order logic," in *Proc. 25th ESOP*, 2016, pp. 503–531.