Cryptographically Secure Information Flow Control on Key-Value Stores

Lucas Waye  
Harvard University  
Cambridge, Massachusetts  
lwayne@seas.harvard.edu

Pablo Buiras  
Harvard University  
Cambridge, Massachusetts  
pbuiras@seas.harvard.edu

Owen Arden  
University of California, Santa Cruz  
Santa Cruz, California  
owen@soe.ucsc.edu

Alejandro Russo  
Chalmers University of Technology  
Gothenburg, Sweden  
russo@chalmers.se

Stephen Chong  
Harvard University  
Cambridge, Massachusetts  
chong@seas.harvard.edu

Abstract
We present Clio, an information flow control (IFC) system that transparently incorporates cryptography to enforce confidentiality and integrity policies on untrusted storage. Clio insulates developers from explicitly manipulating keys and cryptographic primitives by leveraging the policy language of the IFC system to automatically use the appropriate keys and correct cryptographic operations. We prove that Clio is secure with a novel proof technique that is based on a proof style from cryptography together with standard programming languages results. We present a prototype Clio implementation and a case study that demonstrates Clio's practicality.

CCS Concepts
• Security and privacy → Information flow control; Key management; Digital signatures; Public key encryption;

Keywords
information-flow control, cryptography

1 Introduction
Cryptography is critical for applications that securely store and transmit data. It enables the authentication of remote hosts, authorization of privileged operations, and the preservation of confidentiality and integrity of data. However, applying cryptography is a subtle task, often involving setting up configuration options and low-level details that users must get right; even small mistakes can lead to major vulnerabilities [37, 48]. A common approach to address this problem is to raise the level of abstraction. For example, many libraries provide high-level interfaces for establishing TLS [19] network connections (e.g., OpenSSL1) that are very similar to the interfaces for establishing unencrypted connections. These

1Work done while author was at Harvard University.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

CCL '17, October 30-November 3, 2017, Dallas, TX, USA
© 2017 Copyright held by the owner/author(s).
https://doi.org/10.1145/3133956.3134036

libraries are useful (and popular) because they abstract many configuration details, but they also make several assumptions about certificate authorities, valid protocols, and client authentication. Due in part to these assumptions, the interfaces are designed for experienced cryptography programmers and as a result can be used incorrectly by non-experts in spite of their high level of abstraction [61]. Indeed, crypto library misuse is a more prevalent security issue than Cross-Site Scripting (XSS) and SQL Injection [57].

Information flow control (IFC) is an attractive approach to building secure applications because it addresses some of these issues. There has been extensive work in developing expressive information flow policy languages [3, 39, 53] that help clarify a programmer’s intent. Furthermore, many semantic guarantees offered by IFC languages are inherently compositional from a security point of view [24, 64]. However, existing IFC languages (e.g., [18, 28, 41, 47, 54, 55, 63]) generally assume that critical components of the system, such as persistent storage, are trustworthy—the components must enforce the policies specified by the language abstraction. This assumption makes most IFC systems a poor fit for many of the use-cases that cryptographic mechanisms are designed for.

It is tempting to extend IFC guarantees to work with untrusted data storage by simply "plugging-in" cryptography. However, the task is not simple: the threat model of an IFC system extended with cryptography differs from both the standard cryptographic threat models and from standard IFC threat models. Unlike most IFC security models, an attacker in this scenario may have low-level abilities to access signatures and ciphertexts of sensitive data, and the ability to deny access to data by corrupting it (e.g., flipping bits in ciphertexts).

Attackers also have indirect access to the private cryptographic keys through the trusted runtime. An attacker may craft and run programs that have access to the system's cryptographic keys in order to trick the system into inappropriately decrypting or signing information. Cryptographic security models often account for the high-level actions of attackers using oracles that mediate what information an active attacker can learn through interactions with the cryptosystem. These oracles abstractly represent implementation artifacts that could be used by the attacker to distinguish ciphertexts. Ensuring that an actual implementation constrains its behavior to that modeled by an oracle is typically left to developers.

An attacker’s actual interactions with a system often extend beyond the semantics of specific cryptographic primitives and into
application-specific runtime behavior such as how a server responds when a message fails to decrypt or a signature cannot be verified. If an attacker can distinguish this behavior, it may provide them with information about secrets. Building real implementations that provide no additional information to attackers beyond that permitted by the security model can be very challenging.

Therefore, to give developers better tools for building secure applications, we need to ensure that system security is not violated by combining attackers’ low-level abilities and their ability to craft their own programs. This requires extending the attacker’s power beyond that typically considered by IFC models, and representing the attacker’s interactions with the system more precisely than typical cryptographic security models.

This paper presents Clio, a programming language that reconciles IFC and cryptography models to provide guarantees on both ephemeral data within Clio applications and persistent data on an untrusted key-value store. Clio extends the IFC-tool LIO [54] with store and fetch operations for interacting with a persistent key-value store. Like LIO, Clio expresses confidentiality and integrity requirements using security labels: flows of information are controlled throughout the execution of programs to ensure the policies represented by the labels are enforced. Clio encrypts and signs data as it leaves the Clio runtime, and decrypts and verifies as it enters the system. These operations are done automatically according to the security labels—thus avoiding both the mishandling of sensitive data and the misuse of cryptographic mechanisms. Because the behavior of the system is fully specified by the semantics of the Clio language, an attacker’s interactions with the system can be characterized precisely. This results in a strong connection between the power of the attacker in our formal security model and in actual Clio programs.

Clio transparently maps security labels to cryptographic keys and leverages the underlying IFC mechanisms to ensure that keys are not misused within the program. Since we consider attackers capable of denying access to information by corrupting data, Clio extends LIO labels with an availability policy that tracks who can deny access to information (i.e., who may corrupt the data).

Figure 1 presents an overview of the Clio threat model. At a high-level, a Clio program may be a malicious program written by the attacker. All interactions between the runtime and the store are visible to the attacker. Only the (trusted) Clio runtime has access to the keys used to protect information from the attacker, but the attacker may have access to other “low” keys. The Clio runtime never exposes keys directly to program code: they are only used implicitly to protect or verify data as it leaves or enters the Clio runtime.

Attackers may also perform low-level fetch and store operations directly on the key-value store. Using these low-level operations, an attacker may corrupt ciphertexts to make them invalid even when it does not possess the signing keys to make valid modifications. We treat these actions as attacks on the availability of data, rather than on its integrity. A low-availability store is vulnerable to availability attacks, and thus should be prevented from storing data that requires high-availability. Clio’s information flow control mechanisms mediate the attacker’s ability to discover new information or modify signed values by interacting with a Clio program through fetchs and stores to a Clio store.

This paper makes the following contributions:

- A formalization of the ideal semantics of Clio, which models its security without cryptography, and a real semantics, which enforces security cryptographically.
- A novel proof technique that combines standard programming language and cryptographic proof techniques. Using this approach, we characterize the interaction between the high-level security guarantees provided by information flow control and the low-level guarantees offered by the cryptographic mechanisms.
- For confidentiality, we have formalized these guarantees as chosen-term attack (CTA) security, an extension of chosen-plaintext attack (CPA) security to systems where an attacker may choose arbitrary Clio programs that encrypt and decrypt information through the Clio runtime. Though CTA security is predicated on the relatively weak guarantees of CPA crypto primitives, CTA security provides stronger guarantees since it applies to the end-to-end flow of information through the system, including the interactions an active, adaptive attacker might use to distinguish ciphertexts.
- For integrity, we have defined leveraged existential forgery, an extension of existential forgery to systems where an attacker may choose and execute a program to produce signed values.
- A prototype Clio implementation in the form of a Haskell library extending LIO. Our prototype system employs the DC labels model [53], previously used in practical systems (e.g., Hails [23] and COWL [55]). Our implementation extends DC-labels with an availability component, which may be applicable to these existing systems as well.

Our approach uses a computational model of cryptography. However, we do not rely on a formal definition of computational noninterference [30]. Instead, we phrase security in terms of an adversary-based game with a definition much closer to standard cryptographic definitions of security such as CPA security [43]. This approach helps to model an active adversary on the store, something that computational noninterference can not easily capture. Furthermore, we incorporate the semantics of Clio programs and potential attacks against them into the security model. This approach captures the

![Figure 1: Clio threat model. Attackers write Clio programs, read from and write to the store, and observe the runtime’s interactions.](image-url)
power of the attacker more precisely than cryptographic models for active attackers like chosen-ciphertext attack (CCA) security [43].

Our CTA model applies a game-based definition of security in a language setting and is a novel aspect of this work. Computational noninterference and related approaches consider attackers that can only provide different secret inputs to the program. Thus a key contribution of our work is capturing the abilities of an active attacker (that can both supply code to execute and directly manipulate the store) in a crypto-style game that goes beyond CPA security and standard IFC guarantees (noninterference, including computational noninterference). Although our results are specific to Clio, we expect our approach to be useful in proving the security of cryptographic extensions of other information flow languages.

The rest of the paper is structured as follows. Section 2 introduces LIO and Section 3 describes the extensions to it in order to interact with an untrusted store. Section 4 describes the computational model of Clio with cryptography, and Section 5 shows the model’s formal security properties. Section 6 describes the prototype implementation of Clio along with a case study. And finally Section 7 discusses related work, and Section 8 concludes.

2 Background

In this section, we describe the programming model of Clio. Clio is based on LIO [54], a dynamic IFC library implemented in Haskell.

LIO uses Haskell features to control how sensitive information is used and to restrict I/O side-effects. In particular, it implements an embedded language and a runtime monitor based on the notion of a monad, an abstract data type that represents sequences of actions (also known as computations) that may perform side-effects. The basic interface of a monad consists in the fundamental operations return and (∝) (read as “bind”). The expression return x denotes a computation that returns the value denoted by x, performing no side-effects. The function (∝) is used to sequence computations. Specifically, t ∝ λx.t’ takes the result produced by t and applies function λx.t’ to it (which allows computation t’ to depend on the value produced by t). In order to be useful, monads are usually extended with additional primitive operations to selectively allow the desired side-effects. The LIO monad is a specific instance of this pattern equipped with IPC-aware operations that enforce security.

LIO, like many dynamic IFC approaches (e.g., [14, 46, 65]), employs a floating label. Security concerns are represented by labels which form a lattice, a partially-ordered (⊑) set with least upper bounds (⊔) and greatest lower bounds (⊒). A runtime monitor maintains as part of its state a distinguished label ℓcut known as the current label. The current label is similar to the program counter (pc) label of static IFC systems (e.g., [41, 49]): it restricts side-effects in the current computation that may compromise the confidentiality or integrity of data. For example, a computation whose current label is secret cannot write to a public location. LIO operations adjust this label when sensitive information enters the program and use it to validate (or reject) outgoing flows.

When an LIO computation with current label ℓcut observes an entity with label l, its current label is increased (if necessary) to the least upper bound of the two labels, written ℓcut ⊔ l. Thus, the current label “floats up” in the security lattice, so that it is always an upper bound on the security levels of information in the computation.

Ground Value: ːv ::= true | false | ⟨⟩ | l | (v, v)

Value: ːv ::= ⟨⟩ | ⟨v, v⟩ | x | λx.t | LIO | out

Term: ːt ::= ːv | ⟨t, t⟩ | t | fix t | if t then t else t

| t1 ⊔ t2 | t1 ⊔ t2 | t1 ⊒ t2
| return t | t ≫ t
| label t t | labelOf t | unlabel t
| getLabel | getClearance | lowerClearance t
| toLabeled t t | (l t)
| store t t | fetch t t

Ground Type: ːr ::= Bool | ⟨⟩ | Label | (t, r)

Type: ːr ::= ːr | (t, r) | r → r | LIO r | Labeled r

Figure 2: Syntax for Clio values, terms, and types.

Similarly, before performing a side-effect visible to label l, LIO ensures the current label flows to ℓ (ℓcut ⊔ l).

Once the current label within a given computation is raised, it can never be lowered. This can be very restrictive, since, for example, as soon as confidential data is accessed by a computation, the computation will be unable to output any public data. To address this limitation, the toLabeled operation allows evaluation of an LIO computation m in a separate compartment: toLabeled m will run m to completion, and produce a labeled value ov : l, where v is the result of computation m, and l is an over-approximation of the final current label of m. Note that the current label of the enclosing computation is not affected by executing toLabeled m. In general, given a labeled value ov : l, label l is an upper bound on the information conveyed by v. Labeled values can also be created from raw values using operation label, and a labeled value can be read into the current scope with operation unlabel. Creating a labeled value with label l can be regarded as writing into a channel at security level l. Similarly, observing (i.e., unlabeling) a labeled value at l is analogous to reading from a channel at l.

LIO security guarantees. LIO provides a termination-insensitive noninterference-based security guarantee [24]. Intuitively, if a program is noninterfering with respect to confidentiality, then the public outputs of a program reveal nothing about the confidential inputs. More precisely, an attacker A that can observe inputs and outputs with confidentiality label at most lA learns nothing about any input to the program with label l such that l ⊑ lA. Similarly, a program is noninterfering for integrity if an attacker that can control untrusted inputs cannot influence trusted outputs.

2.1 Clio

Clio calculus. Clio is formalized as a typed λ-calculus with call-by-name evaluation, in the same style as LIO [54]. Figure 2 gives the syntax of Clio values, terms, and types. In addition to standard λ-calculus features, Clio includes several security-related extensions that mirror those in LIO, and two operations for interacting with the key-value store, namely store and fetch. As those primitives have nontrivial semantics that involve the external storage, we defer their discussion to Section 3. Security labels have type Label and labeled values have type Labeled r. Computation on
returns a labeled value stored in value with label $l$. The operations getLabel and getClearance can be used to retrieve the current label and clearance respectively.

Rules toLabeled and reset deserve special attention. To evaluate toLabeled $l_1 t$, we first check that $l_1$ is a valid target label ($l_{cur} \sqsubseteq l_{clr}$) and then wrap $t$ in a compartment using the special syntactic form $l_{clr} \triangleright t$, recording the current label and clearance at the time of entering toLabeled and the target label of the operation, $l_1$. Evaluation proceeds by reducing $t$ in the context of the compartment to a value of the form $l_{clr} \times CLIO$. Next, the rule reset evaluates the term $l_{cur} \sqsubseteq t$, first checking that the current label flows to the target of the current toLabeled ($l_{cur} \sqsubseteq l_2$). Finally, the compartment is replaced by a normal label operation and the current label and clearance are restored to their saved values.

**DC Labels.** LIO is parametric in the label format, but for the purposes of this paper in Clio we use DC labels [53] with three components to model confidentiality, integrity, and availability policies. A label $(l_c, l_i, l_a)$ represents a policy with confidentiality $l_c$, integrity $l_i$, and availability $l_a$. Information labeled with $(l_c, l_i, l_a)$ can be read by $l_c$, is vouched for by $l_i$, and is hosted by $l_a$.


dc \triangleright \begin{align*}
    \text{Confidentiality} & : C(l_c), \ I(l_i), \ A(l_a) \\
    \text{Integrity} & : C(l_c), \ I(l_i), \ A(l_a) \\
    \text{Availability} & : C(l_c), \ I(l_i), \ A(l_a)
\end{align*}

dc

Figure 3: Clio language semantics (selected rules).

labeled values occur in the Clio monad using the return and (for) monadic operators. The nonterminals $\text{return } t$ and $\text{return } t' \triangleright t$ are only generated by intermediate reduction steps and are not valid source-level syntax. For convenience, we also distinguish values that can be easily serialized as ground values, $v$. Ground values are all values except functions and Clio computations. To facilitate our extension of LIO with cryptography, we require labeled values to contain only ground values.

Static type checking is performed in the standard way. We elide the typing rules $\triangleright t : \tau$ since they are mostly standard. LIO enforces information flow control dynamically, so it does not rely on its type system to provide security guarantees.

The semantics is given by a small-step reduction relation $\rightarrow$ over Clio configurations (Figure 3).

3 Interacting with an Untrusted Store

Clio extends LIO with a key-value store. The language is extended with two new commands: store $t_k t_v$ puts a labeled value $t_v$ in the store indexed by key $t_k$; fetch $\tau t_k t_v$ command fetches the entry with key $t_k$ if it cannot be fetched, returns the labeled value $t_v$. In both commands, $t_k$ must evaluate to a ground value and the labeled value $t_v$ must evaluate to a labeled ground value with type $\tau$.

\footnote{Complete definitions given in the technical report [59].}
\footnote{The rest can be found in the technical report [59].}
store (possibly overwriting a previous value) should not be influenced by information below the integrity requirements of the store. For availability, the information should not be derived from less available sources than the store’s availability level.

Additionally, we require the current label to flow to \( l \), the label of the value that is being stored (i.e., \( l_{\text{cur}} \leq l \)). Intuitively, this is because an entity that learns the labeled value also learns that the labeled value was put in the store. Current label \( l_{\text{cur}} \) is an upper bound on the information that led to the decision to perform the store, and \( l_{\text{cur}} \) bounds who may learn the labeled value.

For command \( \xrightarrow{\alpha} \), labeled value \( \langle \ell; l_d \rangle \) serves double duty. First, if the store cannot return a suitable value (e.g., because there is no value indexed by key \( \underline{v}_k \), or because cryptographic signature verification fails), then the fetch command evaluates to the default labeled value \( \langle \ell; l_d \rangle \) (which might be an error value or a suitable default). Second, label \( l_d \) specifies an upper bound on the label of any value that may be returned: if the store wants to return a labeled value \( \langle \ell; l \rangle \) where \( l \not\succeq l_d \), then the fetch command evaluates to \( \langle \ell; l_d \rangle \) instead. This allows programmers to specify bounds on information they are willing to read from the store.

Rule \text{Fetch-Valid} is used when a labeled value is successfully fetched from the store. Store event \( \xrightarrow{\alpha} \) at \( \underline{v}_k \) indicates that the store was able to return labeled value \( \langle \ell; l_d \rangle \) indexed by the key \( \underline{v}_k \). Rule \text{Fetch-Invalid} is used when a labeled value cannot be found indexed at the index requested or it does not safely flow to the default labeled value (i.e., it is too secret, too untrustworthy or not available enough), and causes the fetch to evaluate to the specified default labeled value. Since the label of the default value \( l_d \) will be used for the label of the fetched value in general, the availability of the store level should be bounded above by the availability of the label of the default value (i.e., \( \text{A}(\ell) \leq \text{A}(l_d) \)) in both rules, as the label of the fetched value should reflect the fact that anyone from the store could have corrupted the value.

### 3.1 Ideal Store Behavior

We informally describe the ideal behavior of an untrusted store from the perspective of a \text{Clio} program. The ideal store semantics provides a specification of the behavior that a real implementation should strive for, and allows the programmer to focus on functionality and security properties of the store rather than the details of cryptographic enforcement of labeled values. In Section 4 we describe how we use cryptography to achieve this ideal specification.

We use a small step relation \((c, \sigma) \rightarrow (c', \sigma')\) where \((c, \sigma)\) and \((c', \sigma')\) are pairs of a \text{Clio} configuration \(c\) and an ideal store \(\sigma\). An ideal store \(\sigma\) maps ground values \(\underline{v}_k\) to labeled ground values \(\langle \ell; l \rangle\). If a store doesn’t contain a mapping for an index \(\underline{v}_k\), we represent that as mapping it to the distinguished value \(\bot\).

Store events are used to communicate with the store. When a put \(\underline{v}_k \vdash \langle \ell; l \rangle\) at \(\underline{v}_k\) event is emitted, the store is updated appropriately. When the \text{Clio} computation issues a fetch command, the store provides the appropriate event (i.e., either provides value \nothing or \(\xrightarrow{\alpha} \) at \(\underline{v}_k\) for an appropriate labeled value \(\langle \ell; l \rangle\)). For \text{Clio} computation steps that do not interact with the store, store event skip is emitted, and the store is not updated.

\[\text{store: } l_{\text{cur}} \leq \ell \quad l_{\text{cur}} \leq l_1 \quad a = \text{put} \underline{v}_k \vdash \langle \ell; l_1 \rangle \text{ at } \underline{v}_k\]

\[\text{fetch-valid: } \langle l_{\text{cur}}, l_{\text{ch}} \mid \text{store } \underline{v}_k \vdash \langle \ell; l_1 \rangle \rangle \xrightarrow{\alpha} (l_{\text{cur}}, l_{\text{ch}} \mid \text{return } \langle \ell; l_1 \rangle)\]

\[\text{fetch-invalid: } \langle l_{\text{cur}}, l_{\text{ch}} \mid \text{store } \underline{v}_k \vdash \langle \ell; l_1 \rangle \rangle \xrightarrow{\alpha} (l_{\text{cur}}, l_{\text{ch}} \mid \text{return } \langle \ell; l_1 \rangle)\]

**Figure 4:** \text{Clio} language semantics (store and fetch rules).
3.2 Non-Clio Interaction: Threat Model

We assume that programs other than Clio computations may interact with the store and may try to actively or passively subvert the security of Clio programs. Our threat model for these adversarial programs is as follows (and uses store level \( \ell \) to characterize some of the adversaries’ abilities).

- All indices of the key-value store are public information, and an adversary can probe any index of the store and thus notice any and all updates to the store.
- An adversary can read labeled values \( \langle \mu; l_i \rangle \) in the store where the confidentiality level of label \( l_i \) is at least as confidential as the store level \( \ell \) (i.e., \( C(l_i) \sqsubseteq C(\ell) \)).
- An adversary can put labeled values \( \langle \mu; l_i \rangle \) in the store (with arbitrary ground value index \( \mu_i \)) provided the integrity level of store level \( \ell \) is at least as trustworthy as the integrity of label \( l_i \) (i.e., \( I(\ell) \sqsubseteq I(l_i) \)).

An adversary can adaptively interact with the store. That is, the behavior of the adversary may depend upon (possibly probabilistically) changes the adversary detects or values in the store.

We make the following restrictions on adversaries.

- The adversary does not have access to timing information. That is, it cannot observe the time between updates to the store. We defer to orthogonal techniques to mitigate the impact of timing channels [8]. For example, Clio could generate store events on a fixed schedule.
- The adversary cannot observe termination of a Clio program, including abnormal termination due to a failed label check. This assumption can be satisfied by requiring that all Clio programs do not diverge and are checked to ensure normal termination, e.g., by requiring getLabel checks on the label of a labeled value before unlabeling it. Static program analysis can ensure these conditions, and in the rest of the paper we consider only Clio programs that terminate normally.

Note that even though the adversary might have compromised the Clio program, it can only interact with it at runtime through the store. The adversary does not automatically learn everything that the program learns, because data in the Clio runtime is still subject to Clio semantics and the IFC enforcement, which might prevent exfiltration to the store. The Clio semantics thus gives a more precise characterization of the power of the adversary. Rather than proving the security in the presence of a decryption oracle (e.g., CCA or CCA-2 [43]), the Clio runtime prevents system interactions from being used as a decryption oracle, by construction.

We formally model the non-Clio interactions with the store using sequences of adversary interactions \( I \), given in Figure 5. Adversary interactions are skip, put \( \langle \mu; l_i \rangle \), at \( \mu' \) and corrupt \( \mu_1, \ldots, \mu_n \), which, respectively: do nothing; put a labeled value in the store; and delete the mappings for entries at indices \( \mu_1 \) to \( \mu_n \). For storing labeled values, we restrict the integrity of the labeled value stored by non-Clio interactions to be at most at the store level. Sequences of interactions \( I_1 \cdot \ldots \cdot I_n \) are notated as \( \bar{I} \).

To model the adversary actively updating the store, we define a step semantics \( \triangleright \) that includes adversary interactions \( \bar{I} \). We restrict interactions to occur only at low steps, i.e., when the current label of the Clio computation is less than or equal to the store level \( \ell \). (By contrast, a high step is when the current label can not flow to \( \ell \).) This captures the threat model assumption that the attacker cannot observe timing. Rules Low-to-Step and Low-to-High-to-Low-Step in Figure 5 express adversary interactions occurring only at low steps.

4 Realizing Clio

In this section we describe how Clio uses cryptography to enforce the policies on the labeled values through a formal model, called the real Clio store semantics. This model serves as the basis for establishing strong, formally proven, computational guarantees of the Clio system. We first describe how DC labels are enforced with cryptographic mechanisms (Section 4.1), and then describe the real Clio store semantics (Section 4.2).

4.1 Cryptographic DC Labeled Values

Clio, like many systems, identifies security principals with the public key of a cryptographic key pair, and associates the authority to act as a given principal with possession of the corresponding private key. At a high level, Clio ensures that only those with access to a principal’s private key can access information confidential to that principal and vouch for information on behalf of that principal.

Clio tracks key pairs in a keystore. Formally, a keystore is a mapping \( P : p \rightarrow ((0, 1)^*, \{0, 1\}^*) \), where \( p \) is the principal’s well-known name, and the pair of bit strings contains the public and private keys for the principal. In general, the private key for a principal may not be known—represented by \( \bot \)—which corresponds to not knowing the identity of a principal, but not possessing its authority. Keystores are the basis of authority and identity for Clio computations. We use meta-functions on keystores to describe the authority of a keystore in terms of DC labels.\(^5\) Conceptually, a keystore can access and vouch for any information for a principal for which it has the principal’s private key. Meta-function authorityOf\((P)\) returns a label where each component (confidentiality, integrity, and availability) is the conjunction of all principals for which keystore \( P \) has the private key. We also use the keystore to determine the starting label of a Clio program Start\((P)\) and the least restrictive clearance Chr\((P)\), which are, respectively, the most public, trusted, and available label possible and the most confidential, least trusted.

\(^5\)Complete definitions for these functions are in the technical report [59].
and least available data that the computation can compute on, given the keystore’s authority.

Using the principal keystore as a basis for authority and identity for principals, Clio derives a cryptographic protocol that enforces the security policies of safe information flows defined by DC labels.

In the DC label model, labels are made up of triples of formulas. Formulas are conjunctions of categories $C_1 \land \ldots \land C_n$. Categories are disjunctions of principals $p_1 \lor \ldots \lor p_n$. Any principal in a category can read (for confidentiality) and vouch for (for integrity) information bounded above by the level of the category. We enforce that ability cryptographically by ensuring that only principals in the category have access to the private key for that category. Clio achieves this through the use of category keys.

A category key serves as the cryptographic basis of authority and identity for a category. Category keys are made up of the following components: a category public key that is readable by all principals, a category private key that is only readable by members of the category, and a category key signature that is a signature on the category public key and category private key to prove the category key’s authenticity. Category keys are created lazily by Clio as needed and placed in the store. A category key is created using a randomized meta-function$^6$ parameterized by the keystore.

The generated category private key is encrypted for each member of the category separately using each member principal’s public key. To prevent illegitimate creation of category keys, the encrypted category private key and category public key are together signed using the private key of one of the category members.$^7$ When a category key is created and placed in the store, it can be fetched by anyone but decrypted only by the members of the category. When a Clio computation fetches a category key, it verifies the signature of the category key to ensure that a category member actually vouches for it.$^8$ (Failing to verify the signature would allow an adversary to trick a Clio computation into using a category key that is readable by the adversary.)

A Clio computation encrypts data confidential to a formula $C_1 \land \ldots \land C_n$ by chaining the encryptions of the value. It first encrypts using $C_1$’s category public key and then encrypts the resulting ciphertext for formula $C_2 \land \ldots \land C_n$. This form of layered encryption relies on a canonical ordering of categories; we use a lexicographic ordering of principals to ensure a canonical ordering of encryptions and decryptions.

A Clio computation signs data for a formula by signing the data with each category’s private key and then concatenating the signatures together. Verification succeeds only if every category signature can be verified.

Equipped with a mechanism to encrypt and sign data for DC labels that conceptually respects safe information flows in Clio, we use this mechanism to serialize and deserialize labeled values to the store. Given a labeled ground value $ ⟨\langle i_c, i_l, i_d \rangle ; v ⟩$, the value $g$ is signed according to formula $l_c$. The value and signature are encrypted according to formula $l_c$, and the resulting bitstring is the serialization of the labeled value. Deserialization performs decryption and then verification. If deserialization fails, then Clio treats it like a missing entry, and the fetch command that triggered the deserialization would evaluate to the default labeled value.

**Replay Attacks.** Unfortunately, using just encryption and signatures does not faithfully implement the ideal store semantics: the adversary is able to swap entries in the store, or re-use a previous valid serialization, and thus in a limited way modify high-integrity labeled values in the store. We prevent these attacks by requiring that the encryption of the ground value and signature also includes the index value (i.e., the key used to store the labeled value) and a version number. The real Clio semantics keeps track of the last seen version of a labeled value for each index of the store. When a value is serialized, the version of that index is incremented before being put in the store. When the value is deserialized the version is checked to ensure that the version is not before a previously used version for that index. In a distributed setting, this version counter could be implemented as a vector clock between Clio computations to account for concurrent access to the store. However, for simplicity, we use natural numbers for versions in the real Clio store semantics.

### 4.2 Clio Store Semantics

In this section we describe the real Clio store semantics in terms of a small-step probabilistic relation $\rightarrow_p$. The relation models a step taken from a real Clio configuration to a real Clio configuration with probability $p$. A real Clio configuration is a triple $(c, R, V)$ of a Clio configuration $c$, a distribution of sequences of real interactions with the store $R$, and a version map $V$. The version map tracks version numbers for the store to prevent replay attacks, as described above. For technical reasons, instead of the configuration representing the key-value store as a map, we use the history of store interactions (which includes interactions made both by the Clio computation and the adversary). The sequence of interactions applied to the initial store gives the current store. Because the real Clio store semantics are probabilistic (due to the use of a probabilistic cryptosystem and cryptographic-style probabilistic polynomial-time adversaries), configurations contain distributions over sequences of store interactions.

Real interactions $R$ (and their sequences $\mathcal{R}$) are defined in Figure 7 and are similar to interactions with the ideal store. However, instead of labeled values containing ground values, they contain bitstrings $b$ (expressing the low-level details of the cryptosystem and the ability of the adversary to perform bit-level operations). Additionally, the interaction put $ck$ at $c$ represents storing of a category key. These interactions arise from the serialize metafunction, which may create new category keys. Note that the interaction put $b: l$ at $v_k$ does not need an integrity side condition (as it did in the ideal semantics) in the real semantics since there is no distinction between corruptions and valid store interactions.

We use notation

$$\{ X_1, \ldots, X_n \} \mid X_1 \leftarrow D_1; \ldots; X_n \leftarrow D_n \}$$
to describe the probability distribution over the function \( f \) with inputs of random variables \( X_1, \ldots, X_n \) where \( X_i \) is distributed according to distribution \( D_i \) for \( 1 \leq i \leq n \).

Figure 6 presents the inference rules for \( \rightsquigarrow_p \). Internal steps do not affect the interactions or versions. For storing (rule Store), the version of the entry is incremented using the increment function and the real Clio configuration uses a new distribution of interactions \( R' \) containing the interactions to store the labeled value. The new distribution contains the original interactions (distributed according to the original distribution of interactions) along with a concatenation of labeled ciphertexts and any new category keys (distributed according to the distribution given by serialization function). Note that the label of the stored value is not encrypted as it is public information. The configuration steps with probability 1 as the Store rule will be used for all store operations.

When fetching a labeled value, there are three possible rules that can be used depending on the current state of the store: Fetch-Exists, Fetch-Missing, Fetch-Replay. The premise,

\[(\sigma, p) \in \| R(0) \| R \leftarrow R \| \]

in each of these rules means that store \( \sigma \) has probability \( p \) of being produced (by drawing interaction sequence \( R \) from distribution \( R \) and applying \( R \) to the empty store \( 0 \) to give store \( \sigma \)).

Which rule is used for a fetch operation depends on the state of the store, and so the transitions are given by probability less than one. Rule Fetch-Exists is used when the sequence of interactions drawn produces a store that has a labeled value indexed by \( v_k \) that can be correctly deserialized and whose version is not less than the last version seen at this index. Rule Fetch-Missing is used when the sequence of interactions drawn produces a store that either does not have an entry indexed by \( v_k \) or has an entry that cannot be correctly deserialized. Finally, Fetch-Replay rule is used when the sequences of interactions drawn produce a store where an adversary has attempted to replay an old value: the store low step store semantics.

**Figure 7:** Real Clio Low Step Semantics
is a function from distributions of interactions to distributions of interactions, representing the probabilities of interactions an active adversary would perform. Before each low step, the strategy is invoked to produce a distribution of interactions that will affect the store that the Clio computation is using.

Strategy $S$ expresses the ability of the attacker to modify the store. The attacker chooses $S$ (and $t$, $v_0$, and $v_1$), and $S$ interacts with the store during execution. $S$ is a function from (distributions of) interaction sequences to (distributions of) interaction sequences, i.e., a function from a history of what has happened to the store so far to the attacker’s next modifications to the store. Note that we do not explicitly model fetching from the store as an adversary interaction. There is no need for $S$ to fetch to determine the next modification to the store since $S$ effectively observes the entire history of store interactions. At the end of the game when the adversary continuation ($\mathcal{A}_2$) needs to pick $v_0$ or $v_1$, it observes the history of interactions with the store via the interaction sequence $\mathcal{F}_b$, and thus does not need to explicitly get or fetch values.

5 Formal Properties

5.1 Indistinguishability

A cryptosystem is semantically secure if, informally, ciphertexts of messages of equal length are computationally indistinguishable. Two sequences of probability distributions are computationally indistinguishable (written $\{X_n\}_n \approx \{Y_n\}_n$) if for all non-uniform probabilistic polynomial time (ppt) algorithms $\mathcal{A}$,

$$\Pr[\mathcal{A}(x) = 1 \mid x \leftarrow X_n] - \Pr[\mathcal{A}(y) = 1 \mid y \leftarrow Y_n]$$

is negligible in $n$ [26].

In modern cryptosystems, semantic security is defined as indistinguishability under chosen-plaintext attacks (CPA) [43].

**Definition 5.1 (Indistinguishability under Chosen-Plaintext Attack).** Let the random variable $\text{IND}_b(\mathcal{A}, n)$ denote the output of the experiment, where $\mathcal{A}$ is non-uniform ppt, $n \in \mathbb{N}$, $b \in \{0, 1\}:

$$\text{IND}_b(\mathcal{A}, n) = (pk, sk) \leftarrow \text{Gen}(1^n);$$

$$m_0, m_1, \mathcal{A}_2 \leftarrow \text{Enc}(pk, m_b) \text{ s.t. } |m_0| = |m_1|;$$

$$c \leftarrow \text{Enc}(pk, m_b);$$

Output $\mathcal{A}(c)$

$$\Pi = (\text{Gen}, \text{Enc}, \text{Dec}) \text{ is Chosen-Plaintext Attack (CPA) secure if for all non-uniform ppt } \mathcal{A}:$$

$$\{ \text{IND}_0(\mathcal{A}, n) \}_n \approx \{ \text{IND}_1(\mathcal{A}, n) \}_n$$

This definition of indistinguishability phrases the security of the cryptosystem in terms of a game where an adversary receives the public key and then produces two plaintext messages of equal length. One of the two messages is encrypted and the resulting ciphertext given to the adversary. The cryptosystem is CPA Secure if no adversary exists that can produce substantially different distributions of output based on the choice of message. In other words, no computationally-bounded adversary is able to effectively distinguish which message was encrypted.

Clio relies on a semantically secure cryptosystem, but this is insufficient for Clio to protect the confidentiality of secret information. This is because CPA Security provides guarantees only for individually chosen plaintext messages. In contrast, in our setting we consider terms (i.e., programs) chosen by an adversary. There are also many principals and as a result many keys in a real system, so Clio must protect arbitrarily many principals’ information from the adversary. Additionally, the adversary may already have access to some of the keys. Finally, the adversary is active: it can see interactions with the store and issue new interactions adaptively while the program is running. It can attempt to leverage a Clio computation to illegitimately produce a value it should not have, or could try to trick the Clio system into leaking secret information by interacting with the store. Traditionally, these actions of the adversary are modeled by queries to a decryption oracle, as in CCA-2 [43]. Here, they are modeled directly by the Clio language and store semantics.

We chose to formulate a new definition of security that addresses these concerns, as many previous classical definitions of security fall short in this setting:

- Noninterference does not permit the use of computationally secure mechanisms like cryptography.
- CPA security considers only the semantics of the cryptographic algorithms, not the system they are embedded within.
- CCA and CCA2 attempt to model system behavior using oracles, but the connection between these oracles and an actual system is too abstract.

In contrast, we chose to employ a computational model of cryptography that accurately represents the power of the attacker precisely using the semantics of the language and interactions with the store.

With these considerations in mind, we define indistinguishability under a new form of attack: chosen-term attacks (CTA).

**Definition 5.2 (Indistinguishability under Chosen-Term Attack).** Let the random variable $\text{IND}_b(\mathcal{P}, \mathcal{A}, \tilde{p}, j, n)$ denote the output of the following experiment, where $\Pi = (\text{Gen}, \text{Enc}, \text{Dec}, \text{Sign}, \text{Verify})$:

$$\text{IND}_b(\mathcal{P}, \mathcal{A}, \tilde{p}, j, n) = \left(\begin{array}{l}
\mathcal{P} \leftarrow \text{Gen}(\tilde{p}, 1^n); \\
\mathcal{F}_b \leftarrow \text{Pub}(\mathcal{P}); \\
t, v_0, v_1, \mathcal{A}_2 \leftarrow \mathcal{A}(\mathcal{F}_b) \text{ such that } v_0 = \mathcal{F}_b \text{ and } t = \text{Labeled } \tau \rightarrow \text{LIO } \tau' \\
\mathcal{F}_b \leftarrow \mathcal{F}_b \text{ and } v_0 = \text{Labeled } \tau \\
\mathcal{F}_b \leftarrow \mathcal{F}_b \text{ and } v_1 = \text{Labeled } \tau \\
\mathcal{F}_b \leftarrow \text{authorityOf}(\mathcal{F}_b); \\
(c, \mathcal{R}_b, \mathcal{V}') \leftarrow \text{step}_b(\text{Start}(\mathcal{P}), \text{Clr}(\mathcal{P}) \oplus \langle t, v_b \rangle, \mathcal{S}, \mathcal{J}); \\
\mathcal{R}_b \leftarrow \mathcal{R}_b; \text{ Output } \mathcal{A}_2(\mathcal{F}_b)
\right)$$

Clio using $\Pi$ is CTA Secure if for all non-uniform ppt $\mathcal{A}, j \in \mathbb{N}$, keystores $\mathcal{P}$, and principals $\tilde{p}$:

$$\{ \text{IND}_0(\mathcal{P}, \mathcal{A}, \tilde{p}, j, n) \}_n \approx \{ \text{IND}_1(\mathcal{P}, \mathcal{A}, \tilde{p}, j, n) \}_n$$

The CTA game follows the same structure as the CPA game. In addition, we allow the adversary to know certain information (by fixing it in the game), including some part of the keystore ($\mathcal{F}_b$), the set of principals that Clio is protecting ($\tilde{p}$), and the number of low steps the program takes ($j$). Cryptosystem $\Pi$ is used implicitly in the CTA game to generate keys, encrypt, decrypt, sign and verify.$^3$

---

$^3$More formally, $\text{IND}_b, \mathcal{A}$, and the semantics are also parameterized on $\Pi$, and the uses of $\text{Gen}, \text{Enc}, \text{Dec}, \text{Sign}, \text{Verify}$ should be explicitly taken from the tuple $\Pi$ though we elide their explicit usage in our notation for clarity.
In this game setup, Gen(\(\hat{p}, 1^n\)) generates a new keystore \(\mathcal{P}'\) containing private keys for each of the principals in \(\hat{p}\), using the underlying cryptosystem’s Gen function for each keypair. Then, the adversary receives all public keys of the keystore pub(\(\mathcal{P}\)) and returns three well-typed \(\text{Clio}\) terms: a function \(t\), and two program inputs to the function \(v_0\) and \(v_1\) that must be confidentiality-only low equivalent \(\equiv\) (i.e., they may differ only on secret values)\(^{10}\). It also returns a strategy \(S\) that models the behavior of the adversary on the store while the computation is running. Note that the strategy is also polynomial in the security parameter as it is constructed from a non-uniform polynomial time algorithm. The program \(t\) is run with one of the inputs \(v_0\) or \(v_1\) for a fixed number of steps \(j\). The adversary receives the interactions resulting from a run of the program and needs to use that information to determine which secret input the program was run with.

Being secure under a chosen-term attack means that the sequences of interactions between two low-equivalent programs are indistinguishable and hence an adversary does not learn any secret information from the store despite actively interacting with it while the program it chose is running. Note that the adversary receives the full trace of interactions on the store (including its own interactions); this gives it enough information to reconstruct the final state of the store and any intermediate state. For any set of principals, and any adversary store level, the interactions with the store contain no efficiently extractable secret information for all well-typed terminating programs.

**Theorem 5.3 (CTA Security).** If \(\Pi\) is CPA Secure, then \(\text{Clio}\) using \(\Pi\) is CTA Secure.

We prove this theorem in part by induction over the low step relation \(\bowtie_p\), to show that two low equivalent configurations will produce low equivalent configurations, including computationally indistinguishable distributions over sequences of interactions. A subtlety is that we must strengthen the inductive hypothesis to show that sequences of interactions satisfy a stronger syntactic relation (rather than being just computationally indistinguishable).

More concretely, the proof follows three high-level steps. First, we show how a relation \(\bowtie\) on families of distributions of sequences of interactions preserves computational indistinguishability. That is, if \(R_1 \bowtie R_2\) and \(\Pi\) is CPA secure, then \(R_1 = R_2\). Second, we show that as two low equivalent configurations step using the low step relation \(\bowtie_p\), low equivalence is preserved and the interactions they produce satisfy the relation \(\bowtie\). Third, we show that the use of the step metafunction on two low equivalent configurations will produce computationally indistinguishable distributions over distributions of sequences of interactions. Each step of the proof relies on the previous step and the first step relies on the underlying assumptions on the cryptosystem. We now describe each step of the proof in more detail.

**Step 1: Interactions Relation.** We consider pairs of arbitrary distributions of sequences of interactions and show that, if they are both of a certain syntactic form then they are indistinguishable. Importantly, the indistinguishability lemmas do not refer to the \(\text{Clio}\) store semantics, i.e., they merely describe the form of arbitrary interactions that may or may not have come from \(\text{Clio}\). The invariants on pairs of indistinguishable distributions of interactions implicitly require low equivalence of the programs that generated them, and low equivalence circularly requires indistinguishable distributions of interactions. As a result, we describe the lemmas free from the \(\text{Clio}\) store semantics to break the circularity.

We progressively define the relation \(\bowtie\) on a pair of interactions. Initially, distributions of interactions only contain secret encryptions so that we can appeal to a standard cryptographic argument of multi-message security. Formally, for all keystores \(\mathcal{P}_0\), and \(l_1, \ldots, l_k\), such that \(C(l_i) \equiv C(\text{authorityOf}(\mathcal{P}_i))\), and for all \(m_1^{\alpha}, m_2^{\alpha}\) and all principals \(\hat{p}\), if \(|m_1^x| = |m_2^x|\) for all \(1 \leq i \leq k\) and \(\Pi\) is CPA Secure, then

\[
\{ \text{put } b_1^1 : l^1, \text{ at } v_1^1 \cdot \ldots \cdot \text{put } b_k^1 : l^k, \text{ at } v_k^1 \} \quad \mathcal{P} \leftarrow \text{Gen}(\mathcal{I}^n); \\
\{ \text{put } b_1^2 : l^1, \text{ at } v_1^2 \cdot \ldots \cdot \text{put } b_k^2 : l^k, \text{ at } v_k^2 \} \quad \mathcal{P} \leftarrow \text{Gen}(\mathcal{I}^n); \\
\{ \text{(pk}, s^k) \in \text{rng}(\mathcal{P}); \quad b_1^1 \leftarrow \text{Enc}(pk, m_1^1); \quad 1 \leq i \leq k \} \quad \Pi
\]

Using multi-message security as a basis for indistinguishability, we then expand the relation to contain readable encryptions (i.e., ones for which the adversary has the private key to decrypt) where the values encrypted are the same. In the complete definition of \(\bowtie\), we expand it to also contain interactions from a strategy, forming the final relationship on interactions captured by the \(\bowtie\) relation.

We establish an invariant that must hold between pairs in the relation in order for them to be indistinguishable. For example, in the first definition, the lengths of each corresponding message between the pair must be the same. Each intermediate definition of \(\bowtie\) is used to show that a ppt can simulate the extra information in the more generalized definition (thus providing no distinguishing power). For the first definition of the relation containing only secret encryptions, a hybrid argument is used similar to showing multi-message CPA security \([43]\).

**Step 2: Preservation of Low Equivalence.** We show that as two low equivalent programs \(t\) and \(t'\) progress, they simultaneously preserve low equivalence \(t =^C t'\) and the distributions of sequences of interactions they produce \(R\) and \(R'\) are in the relation \(\bowtie\).

We first show that if \(c_0 \xrightarrow{\alpha} c_0'\) and \(c_1 \xrightarrow{\alpha} c_1'\) and \(c_0 =^C c_1\) then \(c_0' =^C c_1'\). This proof takes advantage of the low equivalence preservation proofs for \(\text{LIO}\) in all cases except for the storing and fetching rules. For store events, since all values being stored will have the same type (due to type soundness), and will be ground values, serialized values will have the same message lengths.

We then show that if

\[
(M_0, V_0) \xrightarrow{\alpha} (M_0', V_0')
\]

and

\[
(M_1, V_1) \xrightarrow{\alpha} (M_1', V_1')
\]

and

\[
(M_0 =^C M_1) \land (V_0 = V_1) \land (M_0 \bowtie M_1)
\]

\[10\text{Complete definition of low equivalence is in the technical report \([59]\).} \]
where the adversary is a non-uniform ppt in the size of the key.

that the interactions from the resulting two distributions (i.e., the resulting distribution over distributions of configurations after one step) are computationally indistinguishable.

Figure 8: Low equivalence is preserved in step₁ for two low equivalent configurations c₁ and c₂ and a strategy S.

Figure 8: Low equivalence is preserved in step₁ for two low equivalent configurations c₁ and c₂ and a strategy S.

The proof on ⋆₁ relies on the previous preservation proof on ⋆→ and the indistinguishability results on ⋆.

Step 3: Indistinguishability of the step metafunction. We show that the step metafunction preserves low equivalence. More formally, we show that if c₀ ⋆₁ c₁ and V₀ = V₁ and R₀ × R₁ then

\[ \{⟨R₀', p₀ \cdots p⟩ | ⟨c₀, R₀, V₀⟩ ⋆₁ \cdots p ⋆₁ ⟨c₀', R₀', V₀⟩\} \approx \{⟨R₁', p₀' \cdots p'⟩ | ⟨c₁, R₁, V₁⟩ ⋆₁ \cdots p' ⋆₁ ⟨c₁', R₁', V₁⟩\} \]

Then, the proof proceeds by showing that the probabilities of traces taken by two low equivalent configurations are equal with all but negligible probability. As an example, Figure 8 shows graphically how one step of the trace is handled. We examine the results of step₁ c₁, S, 1) and step₁ c₂, S, 1) where c₁ ⋆₁ c₂. (Note that this setup matches the instantiation of the CTA game where j = 1.) The left rectangle shows the resulting distribution over distributions of configurations after one step of the c₁ configuration. The right circle shows the resulting distribution over distributions of configurations after one step of the c₂ configuration. Due to the results from Step 2, we can reason that c₁' ⋆₁ c₂' and that c₁'' ⋆₁ c₂'' . We can also conclude that R₁ × R₂ and that R₁'' × R₂''. The final step of the proof is to show that the interactions from the resulting two distributions (i.e., the top circle and bottom circle) are computationally indistinguishable. That is, we show that p₁ is equal to p₂ and also p₁' is equal to p₂' with all but negligible probability.

5.2 Leveraged Forgery

Whereas in the previous subsection we considered the security of encryptions, in this case we consider the security of the signatures. We show that an adversary cannot leverage a Clío computation to illegitimately produce a signed value.

A digital signature scheme is secure if it is difficult to forge signatures of messages. Clío requires its digital signature scheme to be secure against existential forgery under a chosen-message attack, where the adversary is a non-uniform ppt in the size of the key. Often stated informally in the literature [25], a digital signature scheme is secure against existential forgery if no adversary can succeed in forging the signature of one message, not necessarily of his choice. Further, the scheme is secure under a chosen-message attack if the adversary is allowed to ask the signer to sign a number of messages of the adversary’s choice. The choice of these messages may depend on previously obtained signatures.

Parallel to CPA and CTA, we adapt the definition of existential forgery for Clío, which we call leveraged forgery. Intuitively, it should not be the case that a high integrity signature can be produced for a value when it is influenced by low integrity information. We capture this intuition in the following theorem:

**Theorem 5.4 (Leveraged Forgery).** For a principal p and all keystores P₀, non-uniform ppts A, and labels l₁, integers j, j', where ℓ = authorityOf(P₀) and l₁ ⊆ ℓ p, if p is secure against existential forgery under chosen-message attacks, then

\[
\Pr \left( \begin{array}{l}
    \{b : l₁ \in \text{Values}_p(\overrightarrow{R'}) \text{ and } b : l₁ \notin \text{Values}_p(\overrightarrow{R})
    \\
    \{p' \leftarrow \text{Gen}(\langle p \rangle, 1^\ell) \text{; } P' = P₀ \uplus P';
    \\
    t, S, A_2 \leftarrow A\left(\text{pub}(P')\right)
    \\
    (c, R, V) \leftarrow \text{step}_t (\langle \text{Start}(P'), \text{Clr}(P') \mid t \rangle, S, j);
    \\
    \overrightarrow{R} \leftarrow R;
    \\
    t', S' \leftarrow A_2(\overrightarrow{R});
    \\
    (c', R', V') \leftarrow \text{step}_{t'} (\langle \text{Start}(P₀), \text{Clr}(P) \mid t' \rangle, S', j);
    \\
    \overrightarrow{R'} \leftarrow \overrightarrow{R'}
\end{array} \right) > 0.
\]

Intuitively, the game is structured as follows. First, an adversary chooses a term t and strategy S that will be run with high integrity (i.e., Start(P) where P has p’s authority). The adversary sees the interactions R produced by the high integrity computation (which in general will include high integrity signatures).

With that information, the adversary constructs a new term t' and new strategy S' that will be run with low integrity (i.e., Start(P₀)). Note that the strategy may internally encode high integrity signatures learned from the high integrity run that it can place in the store.

The interactions produced by this low integrity computation should not contain any high integrity signatures (i.e., are signed by p). The adversary succeeds if it produces a new valid labeled bitstring b : l₁ that did not exist in the first run. In the experiment, the Valuesₚ metafunction extracts the set of valid labeled bitstrings (i.e., can be deserialized correctly) using the parameterized keystore P to perform the category key decryptions.

The proof of this theorem is in two parts. First we show that the label of a value being stored by a computation is no more trustworthy than the current label of computation. Second, we show that the current label never becomes more trustworthy than the starting label. This means that a low integrity execution (i.e., starting from (Start(P₀), Clr(P) \mid t)) cannot produce a high integrity value (i.e., a labeled value b : l₁ such that l₁ \⊆ ℓ p).

6 Clío in Practice

6.1 Implementation

We implemented a Clío prototype as a Haskell library, in the same style as LIO. Building on the LIO code base, the Clío library has an API for defining and running Clío programs embedded in Haskell.
The library also implements a monitor that oversees the execution of the program and orchestrates three interdependent tasks:

- **Information-flow control** C\texttt{lio} executes the usual LIO IFC enforcement mechanism; in particular, it adjusts the current label and clearance and checks that information flows according to the DC labels lattice.
- **External key-value store** C\texttt{lio} handles all interactions with the store, realized as an external Redis [29] database. This is accomplished by using the hedis [44] Haskell library, which implements a Redis client.
- **Cryptography** C\texttt{lio} takes care of managing and handling cryptographic keys as well as invoking cryptographic operations to protect the security of the principals’ data as it crosses the system boundary into/back from the untrusted store. Instead of implementing our own cryptographic schemes, we leverage the third-party cryptonite [27] library.

C\texttt{lio} uses standard cryptographic schemes to protect the information in the store. In particular, for efficiency reasons we use a hybrid scheme that combines asymmetric cryptography with symmetric encryption. The category keys in the store are encrypted and signed with asymmetric schemes, while the entries stored by C\texttt{lio} programs are encrypted with symmetric encryption and signed with an asymmetric signature scheme.

**Asymmetric cryptography** We use cryptonite’s implementation of RSA, specifically OAEP mode for encryption/decryption and PSS for signing/verification, both with 1024-bit keys and using SHA256 as a hash. We get around the message size limitation by chunking the plaintext and encrypting the chunks separately.

**Symmetric encryption** We use cryptonite’s implementation of AES, specifically AES256 in Counter (CTR) mode for symmetric encryption. We use randomized initialization vectors for each encryption. We can use AESNI if the architecture supports it.

Storing and retrieving category keys and labeled values are implemented as discussed in Section 4.1. The technical report [59] has more details.

**Performance** LIO-style enforcement mechanisms have performed adequately in practice, c.f. Hails [23]. We do not expect combining this with off-the-shelf crypto to introduce more than a constant time overhead for fetching and writing into the store. The only additional concern is the overhead of the category key management protocol, which is proportional to the number of distinct categories and their size. Based on the experience obtained by Jif [41], Fabric [33], and Hails, categories are usually small in number and size. Furthermore, creating category keys incurs a one-time cost which can be amortized over multiple runs and programs.

6.2 Case Study

We have implemented a simple case study to illustrate how our prototype C\texttt{lio} implementation can be used to build an application. In this case, we have built a system that models a tax preparation tool and its interactions with a customer (the taxpayer) and the tax reporting agency, communicating via a shared untrusted store. We model these three components as principals C (the customer), P (the preparer) and IRS (the tax reporting agency). The actions of each of these three principals are modeled as separate C\texttt{lio} computations customerCode, preparerCode and irsCode, respectively.

We assume that the store level \( \ell \) restricts writes to the store in confidential contexts, i.e. \( \ell = (\bot, \top, S) \), where S is the principal running as the store. In this scenario, we consider that the principals involved (C, P and IRS) trust each other and are trying to protect their data from all other principals in the system (i.e., from S).

The customer C initially makes a record with his/her personal information, including his/her name, social security number (SSN), declared income and bank account details, modeled as the type TaxpayerInfo. Figure 9 shows the customer code on the left, modeled as a function that takes this record as an argument, \( t_{\text{p}} \). The first step is to label \( t_{\text{p}} \) with the label \((C \lor P \lor IRS, C, S)\). The confidentiality component is a disjunction of all the principals involved in the interaction, reflecting the fact that the customer trusts both the preparer and the IRS with their data and expects them to be able to read it. A more realistic example would also keep the customer’s personal data confidential (i.e. not readable by the IRS and to some extent by the preparer). However, expressing those flows would require an IFC system with declassification, a feature that we have not included in the current version of C\texttt{lio} since it would introduce additional complexity in our model, and semantic security conditions for such systems are still an active area of research [4, 7, 13]. Without declassification, if IRS was not in the label initially, the IFC mechanism would not allow us to release this data (or anything derived from it) to the IRS at a later time. The integrity component of this label is just C since this data can be vouched for only by the customer at this point, while the availability is trusted since these values haven’t been exposed to (and potentially corrupted by) the adversary in the store yet. The final step of the customer is to store their labeled TaxpayerInfo at key “taxpayer_info” for the preparer to see. Note that in practice this operation creates a category key for \((C \lor P \lor IRS)\), stores it in the database and uses it to encrypt the data, which gets signed by C.

The next step is to run the preparer code, shown in the middle of Figure 9. The preparer starts by fetching the taxpayer data at key “taxpayer_info”, using a default empty record labeled with \( l_{\text{t}} = (P \lor IRS, P \lor C, S) \). The entry in the database is labeled differently with \( l_{\text{r}} = (C \lor P \lor IRS, C, S) \), but the operation succeeds because \( l_{\text{r}} \sqsubseteq l_{\text{t}} \) and the availability in \( l_{\text{r}} \), i.e., it reflects the fact that the adversary S might have corrupted this data. The code then starts a toLabeled sub-computation to securely manipulate the labeled taxpayer record without releasing its current label. In the subcomputation, we unlabel this labeled record and use function prepareTaxes to prepare the tax return. Since we are only concerned with the information-flow aspects of the example, we elide the details of how this function works; our code includes a naive implementation but it would be straightforward to extend it to implement a real-world tax preparation operation. The toLabeled block wraps the result in a labeled value \( r \) with label \( l_{\text{t}} \), the argument to toLabeled. Finally, the preparer stores the labeled tax return \( r \) at key “tax_return”. Note that this operation would fail if we had not used toLabeled, since in that case the current label, raised by the unlabel operation, would not flow to \( \ell \), the label of the adversary.

Figure 9 shows the tax agency code on the right. This code fetches the tax return made by the preparer and stored at key “tax_return”. Analogously to the preparer code, we use the default value of the fetch operation to specify the target label of the
result, namely (IRS, P ∨ C ∨ IRS, S), which in this case is once again more restrictive than what is stored in the database. Thereafter the labeled tax return gets unlaabeled and the information is audited in function verifyReturn, which returns a boolean that represents whether the declaration is correct. In a more realistic application, this auditing would be performed inside a toLabeled block too, but since we are not doing any further store operations we let the current label get raised for simplicity.

These three pieces of code are put together in the main function of the program, which we elide for brevity. This function simply generates suitable keystores for the principals involved (using the Clio library function initializeKeyMapIO) and then runs the code for each principal using the evalClio function.

7 Related Work

Language-based approaches. Combining cryptographic and IFC languages is not new. The Decentralized Label Model (DLM) [39] has been extended with cryptographic operations [17, 22, 51, 56]. These extensions, however, either use only symbolic models of cryptography or provide no security properties for their system.

Models for secure declassification are an active area of research in the IFC community (e.g., [8, 15, 40, 60]). It is less clear, though, how such models compose with cryptographic attacker models. Exploring the interactions between declassification and cryptography is very interesting, but a rigorous treatment of it is beyond the scope of this work.

Cryptographically-masked flows [5] account for covert information-flow channels due to the cryptosystem (e.g., an observer may distinguish different ciphertexts for the same message). However, this approach ignores the probability distributions for ciphertexts, which might compromise security in some scenarios [36]. Laud [31] establishes conditions under which secure programs with cryptographically-masked flows satisfy computational noninterference [30]. Fournet and Rezk [21] describe a language that directly embeds cryptographic primitives and provide a language-based model of correctness, where cryptographic games are encoded in the language itself so that security can range from symbolic correctness to computational noninterference.

Information-flow availability has not been extensively studied. Li et al. [32] discuss the relationship between availability and integrity and state a (termination- and progress-insensitive) noninterference property for availability. Zheng and Myers [66] extend the DLM with availability policies, which express which principals are trusted to make data available. In their setting, availability is, in essence, the integrity of progress [6]: low-integrity inputs should not affect the availability of high-availability outputs. In our work, availability tracks the successful verification of signatures and decryption of ciphertexts, and has analogies with Zheng and Myers' approach.

The problem of conducting proofs of trace-based properties of languages with access to cryptographic operations in a computational setting has been studied before. CoSP [10] is a framework for writing computational soundness proofs of symbolic models. Their approach abstracts details such as message scheduling and corruption models and allows for proofs of preservation of trace properties to be developed in a modular fashion.

Cryptographic approaches. There is much work on how to map principals and access policies to cryptographic keys. Attribute-Based Encryption [12] could be used to protect the confidentiality of data for categories and would avoid the need for category keys when encrypting and decrypting. Ring signatures [45] could be used to protect the integrity of data for categories and would similarly avoid the need for category keys when signing and verifying. We take the approach of using simpler cryptographic primitives as they are more amenable to our proofs. Additionally, as a benefit of taking a language-based approach, Clio's ideal semantics is agnostic to the choice of cryptosystem used. From a user's perspective the underlying cryptographic operations could be swapped out in favor of more efficient cryptosystems without changing the semantics of the system (provided the real semantics was shown separately to provide CTA security and security against leveraged forgery).

There is also work on strengthening the guarantees of existing cryptosystems to protect against more powerful adversaries, e.g., Chosen Ciphertext Attack (CCA) [43] security for adversaries that can observe decryptions of arbitrary ciphertexts. CCA security is needed in systems where an adversary can observe (some of) the effects of decrypting arbitrary ciphertexts. In contrast, Clio's security guarantees are based on a very precise definition of the adversary's power over the system. In particular it captures that an adversary cannot observe anything about the decryptions of confidential values due to IFC mechanisms, since the results of such a decryption would be protected by a label that is more confidential than an adversary would have access to. As a result, Clio requires only a CPA secure cryptosystem to be CTA secure.

Systems. DStar [65] extends decentralized IFC in a distributed system. Every DStar node has an exporter that is responsible for communicating over the network. Exporters also establish the security categories trusted by a node via private/public keys. Fabric [34] is a platform and statically-checked fine-grained IFC language. Fabric supports the secure transfer of data as well as code [2] through, in part, the use of cryptographic mechanisms. In contrast to Fabric, Clio provides coarse-grained IFC and uses DC labels instead of the DLM. In contrast to both DStar and Fabric, this work establishes a formal basis for security of the use of cryptography in the system.

Figure 9: Customer code (left), Preparer code (middle), and IRS code (right)
The lack of a formal proof in both DStar and Fabric is not surprising, given that they target more ambitious and complex scenarios (i.e., decentralized information-flow control for distributed systems).

Remote storage. While data can be stored and fetched cryptographically, information can be still leaked through access patterns. Private Information Retrieval protocols aim to avoid such leaks by hiding queries and answers from a potentially malicious server [16] similar to Clio’s threat model. For performance reasons [42, 50], some approaches rely on a small trusted execution environment provided by hardware [20, 58] that provides the cryptographic support needed to obliviously query the data store [9, 52, 62]. This technique can be seen in oblivious computing [35], online advertising [11], and credit networks [38] for clients which are benign or follow an strict access protocol. If clients are malicious, however, attacker’s code may leak information through access patterns. We force communication with the store to occur in non-sensitive contexts. In addition, our language-based techniques could be extended to require untrusted code to follow an oblivious protocol.

8 Conclusion
Clio is a computationally secure coarse-grained dynamic information-flow control library that uses cryptography to protect the confidentiality and integrity of data. The use of cryptography is hidden from the language operations and is controlled instead through familiar language constructs in an existing IFC library, LIO. Further, we present a novel proof technique that combines standard programming language and cryptographic proof techniques to show the interaction between the high-level security guarantees provided by information flow control and the low-level guarantees offered by the cryptographic mechanisms are secure. We also provide a prototype Clio implementation in the form of a Haskell library extending LIO to evaluate its practicality. We see Clio as a way for programmers that are non-expert cryptographers to use cryptography securely.

Acknowledgments
This material is based upon work supported by the National Science Foundation under Grant No.s 1421770 and 1524052. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This research is also supported by the Air Force Research Laboratory and the Swedish research agencies VR and STINT.

References
A Complete Definitions

A.1 Security Lattice Orderings and Operators

We use three lattices (Confidentiality, Integrity, and Availability) whose domains (named C, I, and A) are formulas of principals in conjunctive normal form. We define an information flow ordering \( \sqsubseteq \) (read as “can flow to”). We give the definitions of each below, where \( \Rightarrow, \land, \lor \) are the usual classical logical connectives:

Confidentiality.

\[
\begin{align*}
C \sqsubseteq^C C' & \iff C' \Rightarrow C \\
C \sqsubseteq^C C' & \iff C \sqsubseteq^C C' \text{ and } C \neq C' \\
C \sqcup^C C' & \iff C \land C' \\
C \sqcap^C C' & \iff C \lor C' \\
\bot_C & \equiv True \\
\top_C & \equiv False
\end{align*}
\]

Integrity.

\[
\begin{align*}
I \sqsubseteq^I I' & \iff I \Rightarrow I' \\
I \sqsubseteq^I I' & \iff I \sqsubseteq^I I' \text{ and } I \neq I' \\
I \sqcup^I I' & \iff I \lor I' \\
I \sqcap^I I' & \iff I \land I' \\
\bot_I & \equiv False \\
\top_I & \equiv True
\end{align*}
\]

Availability.

\[
\begin{align*}
A \sqsubseteq^A A' & \iff A \Rightarrow A' \\
A \sqsubseteq^A A' & \iff A \sqsubseteq^A A' \text{ and } A \neq A' \\
A \sqcup^A A' & \iff A \lor A' \\
A \sqcap^A A' & \iff A \land A' \\
\bot_A & \equiv False \\
\top_A & \equiv True
\end{align*}
\]

We define the security lattice of DC labels as a product lattice of the three individual lattices to form a security lattice whose domain is a triple of principal formulas \( l = \langle C, I, A \rangle \) and whose ordering is based on safe information flows. We also define a trust ordering \( \preceq \) (read as “at least as trustworthy as”):

\[
\begin{align*}
\langle C, I, A \rangle \sqsubseteq \langle C', I', A' \rangle & \iff C \sqsubseteq^C C' \text{ and } I \sqsubseteq^I I' \text{ and } A \sqsubseteq^A A' \\
\langle C, I, A \rangle \sqsubseteq \langle C', I', A' \rangle & \iff C \sqsubseteq^C C' \text{ or } I \sqsubseteq^I I' \text{ or } A \sqsubseteq^A A' \\
\langle C, I, A \rangle \sqsubseteq \langle C', I', A' \rangle & \iff C \sqsubseteq^A C' \text{ and } I \sqsubseteq^I I' \text{ and } A \sqsubseteq^A A' \\
\langle C, I, A \rangle \sqcup \langle C', I', A' \rangle & \iff \langle C \cup^C C', I \cup^I I', A \cup^A A' \rangle \\
\langle C, I, A \rangle \cap \langle C', I', A' \rangle & \iff \langle C \cap^C C', I \cap^I I', A \cap^A A' \rangle \\
\bot & \equiv \langle \bot_C, \bot_I, \bot_A \rangle \\
\top & \equiv \langle \top_C, \top_I, \top_A \rangle
\end{align*}
\]

For convenience of avoiding pattern matching over the components of a label when needing to inspect an individual component, we define the following projection functions:

\[
\begin{align*}
\mathcal{C}(\langle C, I, A \rangle) & = C \\
\mathcal{I}(\langle C, I, A \rangle) & = I \\
\mathcal{A}(\langle C, I, A \rangle) & = A
\end{align*}
\]
A.2 Cryptography Background Definitions

- **Distribution Ensemble**: An ensemble of probability distributions is a sequence \(\{X_n\}_{n \in \mathbb{N}}\) of probability distributions.

- **Negligible Function**: A negligible function is a function \(f(x) : \mathbb{N} \to \mathbb{R}\) such that for every positive integer \(c \in \mathbb{Z}^+\) there exists an integer \(N_c\) such that for all \(x > N_c\):

  \[
  \left| f(x) \right| < \frac{1}{x^c}
  \]

- **Computationally Indistinguishable**: Let \(\{X_n\}_n\) and \(\{Y_n\}_n\) be distribution ensembles. Then we say that they are computationally indistinguishable \(\approx\) if for any non-uniform PPT \(\mathcal{A}\) the following function is negligible in \(n\):

  \[
  \left| \Pr[\mathcal{A}(x) = 1 \mid x \leftarrow X_n] - \Pr[\mathcal{A}(y) = 1 \mid y \leftarrow Y_n] \right|
  \]

- **Hybrid Argument**: Let \(X_1, ..., X_m\) be a sequence of probability distributions, where \(m\) is polynomial in \(n\). Suppose there exists a distinguisher \(D\) that distinguishes \(X_1\) and \(X_m\) with probability \(\epsilon\). Then there exists \(i \in [1, m - 1]\) such that \(D\) distinguishes \(X_i\) and \(X_{i+1}\) with probability \(\frac{\epsilon}{m}\) (Proof uses Triangle Inequality)

- **Cryptosystem**: We consider a asymmetric encryption system and signature scheme \(\Pi = (\text{Gen, Enc, Dec, Sign, Verify})\) such that \(\text{Gen} : 1^n \to (\{0, 1\}^*, \{0, 1\}^*)\) representing the public key and private key to use in the asymmetric cryptographic functions.

- **Correctness of Cryptosystem**: For correctness of our encryption system, we require that if \(p \in \{0, 1\}^*\) and \((pk, sk) \leftarrow \text{Gen}(1^n)\) and \(c = \text{Enc}(pk, p)\) then \(p = \text{Dec}(sk, c)\).

  For correctness of our signature scheme, we require that if \(p \in \{0, 1\}^*\) and \((pk, sk) \leftarrow \text{Gen}(1^n)\) and \(s = \text{Sign}(sk, p)\) then \(1 = \text{Verify}(pk, s)\).

  To simplify the notation while maintaining easy-to-prove security properties, we require that the encryption and signature functions operate on independent parts of the key; that is, they internally use a key derivation function (e.g., encryption uses the first half, and signing uses the second half).

- **Security of Cryptosystem**: For our encryption functions, we assume the cryptosystem is CPA secure, defined as follows \cite{43}. Let the random variable \(\text{IND}_b(\mathcal{A}, n)\) denote the output of the experiment, where \(\mathcal{A}\) is a non-uniform p.p.t., \(n \in \mathbb{N}\), \(b \in \{0, 1\}\):

  \[
  \text{IND}_b(\mathcal{A}, n) = (pk, sk) \leftarrow \text{Gen}(1^n);
  \]

  \[
  m_0, m_1, \mathcal{A}_2 \leftarrow \mathcal{A}(pk) \text{ s.t. } |m_0| = |m_1|;
  \]

  \[
  c \leftarrow \text{Enc}(pk, m_b);
  \]

  Output \(\mathcal{A}_2(c)\)

  Then we say that \(\Pi\) is CPA (Chosen-Plaintext Attack) secure if for all non-uniform p.p.t. \(\mathcal{A}\):

  \[
  \left\{ \text{IND}_b(\mathcal{A}, n) \right\}_n \approx \left\{ \text{IND}_1(\mathcal{A}, n) \right\}_n
  \]

  Note that we consider the cryptographic functions themselves to be public.

  For security of the signature scheme, we assume \(\Pi\) is secure against existential forgery.

- **Indistinguishability Corollary**: The CPA definition may be difficult to understand to some as it is phrased in the form of a game. An alternative definition of security (that is a fairly direct consequence of CPA security) is: For all \(p_0, p_1\), if \(|p_0| = |p_1|\) then,

  \[
  \left\{ \text{Enc}(pk, p_0) \mid (pk, sk) \leftarrow \text{Gen}(1^n) \right\}_n \approx \left\{ \text{Enc}(pk, p_1) \mid (pk, sk) \leftarrow \text{Gen}(1^n) \right\}_n
  \]

  Informally, the results of encrypting of equal-length plaintexts are computationally indistinguishable.

- **Digital Signature Forgery**: We require our digital signature scheme to be secure against existential forgery under a Chosen-Message Attack, where the adversary is a non-uniform p.p.t in the size of the key \cite{25}:

  - **Existential Forgery**: The adversary succeeds in forging the signature of one message, not necessarily of his choice.

  - **Chosen-Message Attack**: The adversary is allowed to ask the signer to sign a number of messages of the adversary's choice. The choice of these messages may depend on previously obtained signatures. For example, one may think of a notary public who signs documents on demand.
A.3 LIO Complete Syntax and Typing Rules

Ground Value:
\[ v ::= \text{true} \mid \text{false} \mid () \mid l \mid (v, v) \]

Value:
\[ v ::= v \mid (v, v) \mid x \mid \lambda x . t \mid t^{\text{LIO}} \mid \langle v : l \rangle \]

Term:
\[ t ::= v \mid (v, t) \mid t \mid \text{fix} t \mid \text{if } t \text{ then } t \text{ else } t \]
\[ \mid t_1 \cup t_2 \mid t_1 \cap t_2 \mid t_1 \subseteq t_2 \]
\[ \mid \text{return } t \mid t \gg eq t \]
\[ \mid \text{label } t \mid \text{labelOf } t \mid \text{unlabel } t \]
\[ \mid \text{getLabel} \mid \text{getClearance} \mid \text{lowerClearance} \]
\[ \mid \text{toLabeled } t \mid t \mid \text{fetch } t \]

Ground Type:
\[ \tau ::= \text{Bool} \mid () \mid (\tau, \tau) \]

Type:
\[ \tau ::= \tau \mid (\tau, \tau) \mid \tau \to \tau \mid \text{LIO } \tau \mid \text{Labeled } \tau \]

| BOOL | UNIT | LABEL | PAIR | Labeled | VAR | \[ \text{bind } t_1 : \text{LIO } t_1 \]
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Gamma \vdash b : \text{Bool} )</td>
<td>( \Gamma \vdash () : () )</td>
<td>( \Gamma \vdash l : \text{Label} )</td>
<td>( \Gamma \vdash (t_1, t_2) : (\tau_1, \tau_2) )</td>
<td>( \Gamma \vdash v : \tau )</td>
<td>( \Gamma \vdash x : \tau )</td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{abstraction} \]
\[ \Gamma[ x \mapsto \tau_1 ] \vdash t_1 : \tau_2 \]
\[ \Gamma \vdash \lambda x . t_1 : \tau_1 \to \tau_2 \]

\[ \text{app} \]
\[ \Gamma \vdash t_1 : \tau_1 \to \tau_1 \]
\[ \Gamma \vdash t_2 : \tau_2 \]
\[ \Gamma \vdash t_1 \, t_2 : \tau_1 \]

\[ \text{fix} \]
\[ \Gamma \vdash (t_1 \to \tau_2) \to \tau_2 \]
\[ \Gamma \vdash \text{fix } t_1 : \tau_1 \to \tau_2 \]

\[ \text{LIO} \]
\[ \Gamma \vdash t : \tau \]
\[ \Gamma \vdash \text{LIO } t : \text{LIO } \tau \]

\[ \text{if} \ t_1 : \text{Bool} \quad \Gamma \vdash t_2 : \tau \quad \Gamma \vdash t_3 : \tau \]
\[ \Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : \tau \]

\[ \text{getLabel} \]
\[ \Gamma \vdash \text{getLabel } : \text{Label} \]

\[ \text{getClearance} \]
\[ \Gamma \vdash \text{getClearance } : \text{Label} \]
\[ \Gamma \vdash \text{lowerClearance } t : \text{LIO } () \]

\[ \text{unlabel} \]
\[ \Gamma \vdash \text{unlabel } t : \text{LIO } \tau \]
\[ \Gamma \vdash \text{toLabeled } t_1 t_2 : \text{Labeled } \tau \]

\[ \text{toLabeled} \]
\[ \Gamma \vdash \text{toLabeled } t_1 t_2 : \text{Labeled } \tau \]

\[ \text{bind} \]
\[ \Gamma \vdash t_1 : \text{LIO } t_1 \]
\[ \Gamma \vdash t_2 : \text{LIO } t_2 \]
\[ \Gamma \vdash t_1 \gg eq t_2 : \text{LIO } t_2 \]

\[ \text{reset} \]
\[ \Gamma \vdash t : \text{LIO } \tau \]
\[ \Gamma \vdash t t_1 t_2 : \text{Labeled } \tau \]

\[ \text{store} \]
\[ \Gamma \vdash \text{store } t_1 t_2 : \text{LIO } () \]

\[ \text{fetch} \]
\[ \Gamma \vdash \text{fetch } t_1 t_2 : \text{LIO } \tau \]
A.4 LIO Remaining Step Rules

The program state is \(c = \langle l_{\text{cur}}, l_{\text{clr}} \mid t \rangle\) where \(l_{\text{cur}}\) is the current label, \(l_{\text{clr}}\) is the current clearance. Computation is modeled as a small-step semantics \(\alpha \rightarrow c'\). We use labels \(\alpha\) to represent interaction with the store.

\[
E ::= \varepsilon \mid E \cdot t \mid \text{if } E \text{ then } t \text{ else } t \mid E \gg t \mid \text{label } E \\
\mid \text{label } \not\in E \mid \text{labelOf } E \mid \text{unlabel } E \\
\mid \text{lowerClearance } E \mid \text{toLabeled } E \cdot t \mid \text{toLabeled } E \mid \text{toLabeled } E \\
\mid \text{store } E \cdot t \mid \text{store } E \cdot v \mid \text{fetch } \tau \cdot E \\
\mid \text{fetch } \tau \cdot v \cdot E
\]

Where \(\ell\) is the store adversary level.

\[
\begin{align*}
\text{APP} & \quad \langle l_{\text{cur}}, l_{\text{clr}} \mid (\lambda x.t_1) t_2 \rangle \rightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid [t_2/x] t_1 \rangle \\
\text{ifTrue} & \quad \langle l_{\text{cur}}, l_{\text{clr}} \mid \text{if true then } t_2 \text{ else } t_1 \rangle \rightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid t_2 \rangle \\
\text{ifFalse} & \quad \langle l_{\text{cur}}, l_{\text{clr}} \mid \text{if false then } t_2 \text{ else } t_3 \rangle \rightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid t_3 \rangle \\
\text{lowerClearance} & \quad l_{\text{cur}} \subseteq l_1 \quad l_1 \subseteq l_{\text{clr}} \quad \langle l_{\text{cur}}, l_{\text{clr}} \mid \text{lowerClearance } l_1 \rangle \rightarrow \langle l_{\text{cur}}, l_1 \mid \text{return } () \rangle \\
\text{LabelOp} & \quad \otimes \in \{\cap, \cup, \subseteq\} \quad \vee = l_1 \otimes l_2 \quad \langle l_{\text{cur}}, l_{\text{clr}} \mid \text{LabelOp } \otimes \vee \rangle \rightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid \vee \rangle \\
\text{Step} & \quad \langle l_{\text{cur}}, l_{\text{clr}} \mid t \rangle \rightarrow \langle l'_{\text{cur}}, l'_{\text{clr}} \mid t' \rangle \\
\end{align*}
\]
## A.5 Complete Low Equivalence Relation

### Terms:

<table>
<thead>
<tr>
<th>Term</th>
<th>Equivalence</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{U}_1 = \mathcal{U}_2$</td>
<td>$\equiv_\ell$</td>
<td>$\mathcal{U}_1 = \mathcal{U}_2$</td>
</tr>
<tr>
<td>$(\mathcal{U}_1, l_1)$</td>
<td>$\equiv_\ell$</td>
<td>$(\mathcal{U}_2, l_2)$</td>
</tr>
<tr>
<td>if $\mathcal{U}_1 = \mathcal{U}_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mathcal{U}_1 = \mathcal{U}_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>where</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\circlearrowleft (l_1) \subseteq C(l')$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>and $\circlearrowleft (l_2) \subseteq A(l')$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $\text{typeof } \mathcal{U}_1 = \text{typeof } \mathcal{U}_2$ otherwise

### Relations:

<table>
<thead>
<tr>
<th>Relation</th>
<th>Equivalence</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(v_1, v_2)$</td>
<td>$\equiv_\ell$</td>
<td>$(v_1', v_2')$</td>
</tr>
<tr>
<td>if $v_1 = \ell v'_1$ and $v_2 = \ell v'_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$t_1 t_2$</td>
<td>$\equiv_\ell$</td>
<td>$t_1' t_2'$</td>
</tr>
<tr>
<td>if $t_1 = \ell t_1'$ and $t_2 = \ell t_2'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fix $t$</td>
<td>$\equiv_\ell$</td>
<td>fix $t'$</td>
</tr>
<tr>
<td>if $t = \ell t'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>if $t_1$ then $t_2$ else $t_3$</td>
<td>$\equiv_\ell$</td>
<td>if $t_1$ then $t_1'$ else $t_3'$</td>
</tr>
<tr>
<td>if $t_1 = \ell t_1'$ and $t_2 = \ell t_2'$ and $t_3 = \ell t_3'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return $t$</td>
<td>$\equiv_\ell$</td>
<td>return $t'$</td>
</tr>
<tr>
<td>if $t = \ell t'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$l_1 \bowtie l_2$</td>
<td>$\equiv_\ell$</td>
<td>$l_1' \bowtie l_2'$</td>
</tr>
<tr>
<td>if $l_1 = \ell l_1'$ and $l_2 = \ell l_2'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>label $t_1 t_2$</td>
<td>$\equiv_\ell$</td>
<td>label $t_1' t_2'$</td>
</tr>
<tr>
<td>if $t_1 = \ell t_1'$ and $t_2 = \ell t_2'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>labelOf $t$</td>
<td>$\equiv_\ell$</td>
<td>labelOf $t'$</td>
</tr>
<tr>
<td>if $t = \ell t'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>unlabel $t$</td>
<td>$\equiv_\ell$</td>
<td>unlabel $t'$</td>
</tr>
<tr>
<td>if $t = \ell t'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getLabel</td>
<td>$\equiv_\ell$</td>
<td>getLabel</td>
</tr>
</tbody>
</table>

### Configurations:

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Equivalence</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(l'<em>{\text{cur}}, l'</em>{\text{clr}}</td>
<td>t)$</td>
<td>$\equiv_\ell$</td>
</tr>
<tr>
<td>if $t = \ell t'$ and $l_{\text{cur}} = l'<em>{\text{cur}}$ and $l</em>{\text{clr}} = l'_{\text{clr}}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$(l'<em>{\text{cur}}, l'</em>{\text{clr}}</td>
<td>t)$</td>
<td>$\equiv_\ell$</td>
</tr>
<tr>
<td>if $(l_{\text{cur}} \in C(l)$ and $(l'<em>{\text{cur}} \in C(l)$ and $(l</em>{\text{clr}} \in C(l)$ and $(l'_{\text{clr}} \in C(l)$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Confidentiality Low Equivalence

Define $\equiv^C_\ell$ to be the low equivalence relation with respect to only confidentiality. The integrity and availability parts of the label are ignored.
A.6 Ideal Clio Complete Syntax and Semantics

The ideal Clio state is \(<c, \sigma>\) where \(c\) is the LIO configuration and \(\sigma\) is a mapping \(\sigma : v \mapsto \langle l, k \rangle \), where \(\perp\) represents a corrupted entry.

Internal-Step

\[
\frac{c \rightarrow c'}{(c, \sigma) \rightsquigarrow (c', \sigma)}
\]

Store

\[
\frac{\text{put } \langle l_1 \rangle \text{ at } v_k}{(c, \sigma) \rightsquigarrow (c', \sigma') \quad \sigma' = \sigma[v_k \mapsto \langle l_1 \rangle]}
\]

Fetch-Exists

\[
\frac{\text{got } \langle l_1 \rangle \text{ at } v_k}{(c, \sigma) \rightsquigarrow (c', \sigma') \quad \langle l_1 \rangle = \sigma(v_k)}
\]

Fetch-Missing

\[
\frac{\text{nothing at } v_k}{(c, \sigma) \rightsquigarrow (c', \sigma') \quad \sigma(v_k) = \perp}
\]

Ideal interactions \(I\) are given by the following syntax:

\[
\mathcal{I} := I \cdot \mathcal{I} \mid I \mid \text{skip} = \lambda \sigma. \sigma \mid \text{put } \langle l_1 \rangle \text{ at } v_k = \lambda \sigma. \sigma[v_k \mapsto \langle l_1 \rangle] \mid \text{corrupt } v_1, \ldots, v_n = \lambda \sigma. \sigma[v_1 \mapsto \perp; \ldots; v_n \mapsto \perp}
\]

The low-step relation \(\bowtie\) from ideal Clio configurations to Clio configurations using adversary interaction \(\mathcal{I}\).

Low-Step

\[
\frac{\langle c, \mathcal{I}(\sigma) \rangle \rightsquigarrow \langle c', \sigma' \rangle \quad \text{PC}(c) \sqsubseteq \ell \quad \text{PC}(c') \sqsubseteq \ell}{((c, \sigma), \mathcal{I}) \bowtie (c', \sigma')}
\]

Low-to-High-to-Low-Step

\[
\frac{\langle c, \mathcal{I}(\sigma) \rangle \rightsquigarrow \langle c_0, \sigma_0 \rangle \quad \forall 0 \leq i < j, \text{PC}(c_i) \not\sqsubset \ell \quad \text{PC}(c_j) \sqsubseteq \ell}{((c, \sigma), \mathcal{I}) \bowtie (c_j, \sigma_j)}
\]
A.7 Complete Definitions for Labeled Value Serialization

- initialize_ckp(σ, C) = (fetch_ckp(σ, C), skip) if fetch_ckp(σ, C) defined
- initialize_ckp(σ, C) = (fetch_ckp(R(σ, C), R) if fetch_ckp(σ, C) undefined and R = create_ckp(σ, C)
- create_ckp(σ, C) = store (p1 ∨ ... ∨ pn) (pk, m, s) where
  (pk, sk) ← Gen(1^s)
  m_0 = 0
  for i from 1 to n:
    (pk_i, sk_i) = P(p_i)
    m_i = m_{i-1}[p_i ↦ Enc(pk_i, sk_i)]
  s ← Sign(sk_i, (pk, m)) if sk_i ≠ ⊥
- fetch_ckp(σ, C) = (fetch_ckp(R(σ, C), R) where
  (pk, m, s) = σ (p1 ∨ ... ∨ pn) = (pk, sk) where
  (pk_i, sk_i) = P(p_i) where i chosen s.t. sk_i ≠ ⊥
  sk = Dec(s_k, m(p_i))
  Verify(pk_i, (pk, m), s) = 1 for some pk_i in the category.
- Enc_C(σ_0, C_1 ∧ ... ∧ C_n, l, v_0) = (R_n · ... · R_1, v_n) where
  for i from 1 to n:
    ((pk_i, sk_i), R_i) ← initialize_ckp(σ_{i-1}, C_i)
    σ_i = R_i(σ_{i-1})
    v_i ← Enc(pk_i, v_{i-1})
- Sign_C(σ_0, C_1 ∧ ... ∧ C_n, v) = (R_n · ... · R_1, s_1, ..., s_n) where
  for i from 1 to n:
    ((pk_i, sk_i), R_i) ← initialize_ckp(σ_{i-1}, C_i)
    σ_i = R_i(σ_{i-1})
    s_i ← Sign(sk_i, v)
- Dec_C(σ, C_1 ∧ ... ∧ C_n, l, v_0) = v_0 where
  for i from n to 1:
    (pk_i, sk_i) = fetch_ckp(σ, C_i)
    v_{i-1} = Dec(sk_i, v_i)
- Verify_C(σ, C_1 ∧ ... ∧ C_n, v, s_1, ..., s_n) = 1 if
  for i from n to 1:
    (pk_i, sk_i) = fetch_ckp(σ, C_i)
    1 = Verify(pk_i, v, s_i)

Similar to the category key meta-functions, we also annotate the results of the meta-functions with the interactions made on the store so that we can track what actions are being taken on the crypto store.

With these cryptographic functions operating on labels, we are now ready to describe the meta-functions which convert a labeled value to a bit string and vice-versa.

- serialize_C = \| (R_n · R_{i-1}, b; l) \| (R, b) ← Enc_C(R_n, l, (v_1, s_1, ..., s_n)); (R_k, s_k, ..., s_n) ← Sign_C(σ, l, v) \|
- deserialize_C = \| (v, l, τ) = v_1 \| if Verify_C(σ, l, v, s_1, ..., s_n) = 1 and Dec_C(σ, l, b) = (v_1, s_1, ..., s_n) and typeOf v = τ

We use the convenience function pub(whatever) to represent the projection of the keystore that only contains the public key parts of the keystores, and no private keys.
A.8 Real CL10 Complete Syntax and Semantics

**Keystores:** \( P : p \rightarrow (b, b_\bot) \)

**Bit strings:** \( b \in \{0, 1\}^* \)

**Stores:** \( \sigma : (v \rightarrow \langle b \cdot l \rangle, \bot) \cup (C \rightarrow c_{k\bot}) \)

**Versions:** \( V : \emptyset \rightarrow n \)

**Interactions:**
\[
R \equiv R \cdot R \mid R
\]
\[
R \equiv \text{skip} = \lambda \sigma. \sigma
\]
\[
\text{put} \; c_{k\bot} \; \text{at} \; C = \lambda \sigma. \sigma[C \mapsto c_{k\bot}]
\]
\[
\text{put} \; \langle b \cdot l \rangle \; \text{at} \; v_k = \lambda \sigma. \sigma[v_k \mapsto \langle b \cdot l \rangle]
\]

**Strategies:** \( S : R \rightarrow R \)

**Category Keys:** \( c_{k\bot} \equiv (b, \; esk, \; b') \)

**Encrypted Keys:** \( esk 

\text{keystore Label Functions}
\]

\[
\text{min}(P) = \bigvee p_1 \vee \cdots \vee p_n \text{ for all } p_i \in \text{dom}(P)
\]

\[
\text{max}(P) = \bigwedge p_1 \wedge \cdots \wedge p_n \text{ for all } p_i \in \text{dom}(P)
\]

\[
\text{Start}(P) = (\text{min}(P), \text{max}(P), \text{max}(P))
\]

\[
\text{Clr}(P) = (\text{max}(P), \text{min}(P), \text{min}(P))
\]

\[
\text{authorityOf}(P) = (\text{max}(P), \text{max}(P), \text{max}(P))
\]

**Interaction:**
\[
R \equiv \text{skip} = \lambda \sigma. \sigma
\]
\[
\text{put} \; \langle b \cdot l \rangle \; \text{at} \; v_k = \lambda \sigma. \sigma[v_k \mapsto \langle b \cdot l \rangle]
\]
\[
\text{put} \; c_{k\bot} \; \text{at} \; C = \lambda \sigma. \sigma[C \mapsto c_{k\bot}]
\]
The interaction concatenation operation \( R \cdot R \) sequences interactions. We use the notation \( \overrightarrow{R} = R \cdot \ldots \cdot R \) to denote a sequence of interactions.

The real \( \text{Clio} \) state is \( \langle c, R, V \rangle \) where \( c \) is the \( \text{LIO} \) configuration, \( \sigma \) is the store.

**Internal-Step**

\[
c \rightarrow c'
\]

\[
\langle c, R, V \rangle \xrightarrow{\text{Internal-Step}} \langle c', R, V \rangle
\]

**Store**

\[
c \xrightarrow{\text{put } (\omega; l_1)} c'
\]

\[
n = \text{increment}(V(\overrightarrow{\nu}_k)) \quad V' = V[\overrightarrow{\nu}_k \mapsto n]
\]

\[
R' = \parallel \text{put } (\omega; l_1) \at \overrightarrow{\nu}_k \cdot \overrightarrow{R} \cdot \overrightarrow{R} \leftarrow R;
\]

\[
(\overrightarrow{R'}, \omega; l_1) \leftarrow \text{serialize}_\sigma(\sigma, ((\nu, \overrightarrow{\nu}_k, n); l_1)) \parallel
\]

\[
\langle c, R, V \rangle \xrightarrow{\text{Internal-Step}} \langle c', R', V' \rangle
\]

**Fetch-Exists**

\[
c \xrightarrow{\text{get } \cdot \omega; l_1} c' \quad n \notin V(\overrightarrow{\nu}_k)
\]

\[
(\sigma, p) \in \parallel \overrightarrow{R}(0) \mid \overrightarrow{R} \leftarrow R \parallel \quad p > 0
\]

\[
(\nu, \overrightarrow{\nu}_k, n); l_1 \leftarrow \text{deserialize}_\sigma(\sigma, \nu, \overrightarrow{\nu}_k, \tau)
\]

\[
\langle c, R, V \rangle \xrightarrow{\text{Fetch-Exists}} \langle c, R, V \rangle
\]

**Fetch-Missing**

\[
c \xrightarrow{\text{nothing } \cdot \omega; l_1} c'
\]

\[
(\sigma, p) \in \parallel \overrightarrow{R}(0) \mid \overrightarrow{R} \leftarrow R \parallel \quad p > 0
\]

\[
\text{deserialize}_\sigma(\sigma, \nu, \overrightarrow{\nu}_k, \tau) \text{ undefined}
\]

\[
\langle c, R, V \rangle \xrightarrow{\text{Fetch-Missing}} \langle c, R, V \rangle
\]

**Fetch-Replay**

\[
c \xrightarrow{\text{nothing } \cdot \omega; l_1} c' \quad \nu' \neq \nu_k \text{ or } n < V(\overrightarrow{\nu}_k)
\]

\[
(\sigma, p) \in \parallel \overrightarrow{R}(0) \mid \overrightarrow{R} \leftarrow R \parallel \quad p > 0
\]

\[
(\nu, \nu', \overrightarrow{\nu}_k, n); l_1 \leftarrow \text{deserialize}_\sigma(\sigma, \nu, \overrightarrow{\nu}_k, \tau)
\]

\[
\langle c, R, V \rangle \xrightarrow{\text{Fetch-Replay}} \langle c, R, V \rangle
\]

The low-step relation \( \prec_p \) from real \( \text{Clio} \) configurations and adversary interactions to \( \text{Clio} \) configurations with probability \( p \).

**Low-Step**

\[
R' = \parallel \overrightarrow{R}_A \cdot \overrightarrow{R} \mid \overrightarrow{R}_A \leftarrow \overrightarrow{R}_A: \overrightarrow{R} \leftarrow R \parallel
\]

\[
\langle c, R', V \rangle \xrightarrow{\text{Low-Step}} \langle c', R'', V' \rangle
\]

\[
\text{PC}(c) \subseteq \mathbb{T}(t) \quad \text{PC}(c') \subseteq \mathbb{T}(t)
\]

\[
\langle c, R, V \rangle \triangleleft \prec_p \langle c', R'', V' \rangle
\]

**Low-to-High-to-Low-Step**

\[
R' = \parallel \overrightarrow{R}_A \cdot \overrightarrow{R} \mid \overrightarrow{R}_A \leftarrow \overrightarrow{R}_A: \overrightarrow{R} \leftarrow R \parallel
\]

\[
\langle c, R', V \rangle \xrightarrow{\text{Low-to-High-to-Low-Step}} \langle c_0, R_0, V_0 \rangle \quad \langle c_0, R', V \rangle \xrightarrow{\text{Low-to-High-to-Low-Step}} \langle c_1, R_1, V \rangle
\]

\[
\forall_{0 \leq i < j}. \text{PC}(c_i) \not\subset \mathbb{T} \quad \text{PC}(c_j) \not\subset \mathbb{T} \quad p = \prod_{0 \leq i < j} p_i
\]

\[
\langle c, R, V \rangle \prec_p \prec_p \langle c_j, R_j, V_j \rangle
\]

The step function encodes the distribution of real \( \text{Clio} \) states after taking \( j \) low steps:

\[
\text{step}^j_p(c_0, S, 1) = \left\{ (c_1, R_1, V_1, p_0 \cdot p_1) \mid (c_0, \{\text{skip } 1\}, V_0, S(\{\text{skip } 1\})) \prec_p \langle c_1, R_1, V_1 \rangle \right\}
\]

\[
\text{step}^j_p(c_0, S, j + 1) = \left\{ (c_2, R_2, V_2, p_0 \cdot p_1) \mid (c_1, R_1, V_1, p_0 \in \text{step}^j_p(c_0, S, j); \quad (c_1, R_1, V_1) \prec_p \langle c_2, R_2, V_2 \rangle \right\}
\]

Note that, we consider only configurations and strategies that can and always will take at least \( j \) low steps for all strategies. That is, there is no possibility for a trace to fail to make a low step before \( j \) low steps. As a result, the step function will always produce a distribution (i.e., their probabilities will add up to 1). The program should be written in such a way that it is defensively written to ensure that it can take at least \( j \) low steps.
B Complete Theorems and Proofs

B.1 Clio Interaction Indistinguishability Lemmas

Lemma B.1 (Round 1: Multi-Message Security). For all \( m^i_{[1,2]} \ldots m^0_{[1,2]} \) and all principals \( \tilde{p} \). if \( |m^i_0| = |m^i_1| \) for all \( 1 \leq i \leq k \), and \( \Pi \) is CPA Secure, then

\[
\left[ \text{Enc}(pk_1^i, m^i_1), \ldots, \text{Enc}(pk_1^i, m^i_1) \right] \overset{\approx}{\leftrightarrow} \left[ \text{Enc}(pk_1^i, m^i_1), \ldots, \text{Enc}(pk_1^i, m^i_1) \right] \quad (1 \leq i \leq k)
\]

\[
\left[ \text{Enc}(pk_2^i, m^i_1), \ldots, \text{Enc}(pk_2^i, m^i_1) \right] \overset{\approx}{\leftrightarrow} \left[ \text{Enc}(pk_2^i, m^i_1), \ldots, \text{Enc}(pk_2^i, m^i_1) \right] \quad (1 \leq i \leq k)
\]

Proof. We perform a proof by contradiction: we assume the consequent does not hold and construct a counter-example to the CPA-security of \( \Pi \).

The lengths of the sequences of encryptions are equal by setup. The sequences are also polynomial in \( n \) as each low step produces a polynomial number of messages and the number of low steps is polynomial in \( n \) as \( k \) is fixed.

This setup is equivalent to the multi-message CPA security problem. We use the same general technique to show that single-message CPA security gives rise to multi-message CPA security by using a hybrid argument.

We can define a hybrid sequence of messages

\[
H_i = \{ m^i_1; \ldots; m^i_1; m^i_{1+1}; \ldots; m^i_k \}
\]

such that at the point \( i \) we switch from using the sequence of messages from the first run to using the sequence of messages from the second run. By the hybrid argument, there must exist an \( i \) that distinguishes \( H_i \) and \( H_{i+1} \) with non-negligible probability in \( n \). We will fix on that \( i \). So we can now construct the following CPA adversary:

1. By assumption of our proof by contradiction, there exists a Round 1 adversary that can distinguish \( H_i \) and \( H_{i+1} \) for a particular \( i \) and particular \( m^i_{[1,1]}, \ldots, m^i_{[k,1]} \), and call it \( A_{R1} \).

2. In our construction, we generate a new keystore \( P \) as defined in the lemma statement and give the plaintext messages \( m^i_{1+1} \) and \( m^i_{1+1} \) to the CPA game and it will then provide us back a ciphertext \( c \) of one of the messages.

3. We create a new sequence of encrypted messages in the following way:

\[
H = \left[ \text{Enc}(pk_1^i, m^i_1); \ldots; \text{Enc}(pk_1^i, m^i_1); c; \text{Enc}(pk_1^{i+2}, m^i_{1+2}); \ldots; \text{Enc}(pk_2^n, m^i_k) \right] \quad (1 \leq o \leq k)
\]

In the case where the CPA game chose message \( m^i_{1+1} \) we have that \( H = H_i \) and in the other case where \( m^i_{1+1} \) we have that \( H = H_{i+1} \). Since \( A_{R1} \) can distinguish exactly this case and that the choice of message encrypted determines which sequence of messages was used, we can as a result distinguish which plain-text message was chosen by the CPA game with non-negligible probability.

As a result of constructing a CPA adversary that can distinguish plain-text messages with non-negligible probability, we have shown a contradiction, and can conclude that the above sequences of encryptions is indistinguishable. 

\[\square\]
Definition B.2 (Low Equivalent Interactions). Let $L_P(R)$ to be a fixed function (i.e., it does not change its behavior based on its inputs) from interactions to interactions such that the result contains the original sequence of interactions with low interactions added at statically fixed locations in the sequence. Let the resulting sequences of interactions be called low equivalent interactions.

A low interaction is a skip command or a put $\langle b: l_1 \rangle$ at $v_k$ such that

$$C(l_1) \subseteq C(\text{authorityOf}(P)) \text{ and } \langle (v, v_k, n): l_1 \rangle = \text{deserialize}_P(R, \langle b: l_1 \rangle) \text{ or put } \tilde{p} \text{ at } C.$$ 

Lemma B.3 (Round 2: Secret and Low Equivalent Interactions). For all keystores $P_0$, and $l_1, \ldots, l_k$, such that $C(l_i) \subseteq C(\text{authorityOf}(P))$, and for all $m_1^{1,2} \ldots m_k^{1,2}$ and all principals $\tilde{p}$, if $|m_i^1| = |m_i^2|$ for all $1 \leq i \leq k$ and $\Pi$ is CPA Secure, then

$$|P \leftarrow \text{Gen}(1^n); (pk_i, sk_i) \in \text{rng}(\Pi); b_i \leftarrow \text{Enc}(pk_i, m_i^1); 1 \leq i \leq k | \approx (|P \leftarrow \text{Gen}(1^n); (pk_i, sk_i) \in \text{rng}(\Pi); b_i \leftarrow \text{Enc}(pk_i, m_i^2); 1 \leq i \leq k |$$

Proof. We perform a reduction to the Round 1 adversary. That is, if there exists a Round 2 adversary $\mathcal{A}_{R2}$ then there also exists a Round 1 adversary, which will provide a contradiction.

We construct our adversary as follows. Given a sequence of secret encryptions (from Round 1), we can construct the input to the $L$ function by constructing a constant set of labels $l_1$ to $l_n$ arbitrarily so long as they flow to $C(\text{authorityOf}(P))$, which is a static property and also arbitrary input keys. We can then also construct the entry keys $v_i$ in the same static fashion. When then just apply the deterministic $L$ function and pass that to $\mathcal{A}_{R2}$.

Since we have shown how to construct a Round 1 adversary from a Round 2 adversary, we have a contradiction of the Round 1 lemma, so we conclude our proof.

$\square$
Definition B.4 (Clio Interactions). Let \( m_1^{1,2}, \ldots, m_k^{1,2} \) be sequences of messages such that \( |m_1^{1,2}| = |m_k^{1,2}| \). Further, let \( \overline{R}_1^{1,2}, \ldots, \overline{R}_k^{1,2} \) be sequences of low equivalent interactions (from Definition 2) whose ciphertexts are based on slices of the underlying message. For example

\[
\overline{R}_1^{1} = \text{put } (b, \{ p \mapsto \text{Enc}(pk, m_1^1) \}) \text{ at } C \cdot \text{put } \text{Enc}(pk, m_1^2) \text{ at } \Sigma_1
\]
\[
\overline{R}_1^{2} = \text{skip}
\]
\[
\overline{R}_1^{3} = \text{put } \text{Enc}(pk', m_1^1) \text{ at } \Sigma_1
\]
\[
\ldots
\]
\[
\overline{R}_1^{j} = \text{put } \text{Enc}(pk'', m_1^1) \text{ at } \Sigma_1
\]

Then we say two sequences of interactions are \( \text{Clio equivalent} \) if they are of the form, with all but negligible probability,

\[
\{ \overline{R}_1^{1}, \ldots, \overline{R}_1^{j} \} \quad \text{iff they are of the form, with all but negligible probability,}
\]

\[
\{ \overline{R}_2^{1}, \ldots, \overline{R}_2^{j} \}
\]

Proof. Similar to the previous rounds, we will reduce this problem to the Round 2 Secret and Low Equivalent Interactions indistinguishability problem. If there exists a Round 3 adversary \( \mathcal{A}_{R3} \) then there also exists a Round 2 adversary, which will provide a contradiction.

We construct our adversary as follows. Given a sequence of low equivalent interactions (from Round 2) \( \overline{R}_1 \) and \( \overline{R}_2 \), we subdivide the interactions into sequences of interactions \( \overline{R}_1^{1,2}, \ldots, \overline{R}_1^{1,2} \). We also add the categories in storing category keys arbitrarily in a static way and also add the category key signatures (as it was not in the previous round) by just performing the signing process according to the \( \text{initialize \_ck} \) function.

For the strategy interactions, we just perform the draws from the strategy starting from the end of the sequences of interactions, working backwards and place them in their corresponding positions, i.e., \( \overline{R}_1^{1} \leftarrow \mathcal{S}(\overline{R}_1^{1} \ldots \overline{R}_1^{j}) \) for \( 1 < s < j \). We note that, although the interactions may differ they are indistinguishable. That is because the first sequence of interactions \( \overline{R}_1^{1,2} \) is a sub-problem of the Round 2 sequences of interactions. As a result, since the two sequences of interactions are computationally indistinguishable, then their corresponding draws from the strategy are also computationally indistinguishable.

As a result, we can then pass the final sequence of interactions to \( \mathcal{A}_{R2} \) to distinguish the distributions. Since we have shown how to construct a Round 2 adversary from a Round 3 adversary, we have a contradiction of the Round 2 lemma, so we conclude our proof.
B.2 CLIO Preservation of Low Equivalence

Lemma B.6 (Preservation of Low Equivalence). For all keystores $P_0$ where $\ell = \text{authorityOf}(P_0)$, LIO configurations $c_1, c_2$, strategies $S$, and principals $p, j \in \mathbb{N}$, if $\Pi$ is CPA Secure and $c_1 \equiv^C_{P} c_2$, then

$$\Pr\left[ (c'_1, R_1, V_1) \not\equiv_{P}^C (c'_2, R_2, V_2) \text{ or } V_1 \not\equiv V_2 \right]$$

is negligible in $n$, and

\[
\begin{array}{c}
\{ R_1 \mid P' \leftarrow \text{Gen}(\tilde{p}, 1^n); P = P_0 \uplus P'; (c'_1, R_1, V_1) \leftarrow \text{step}_{P}^j (c_1, S, j); \\
\{ R_2 \mid P' \leftarrow \text{Gen}(\tilde{p}, 1^n); P = P_0 \uplus P'; (c'_2, R_2, V_2) \leftarrow \text{step}_{P}^j (c_2, S, j); 
\end{array}
\]

is negligible in $n$, and

\[
\begin{array}{c}
\{ R_1 \mid P' \leftarrow \text{Gen}(\tilde{p}, 1^n); P = P_0 \uplus P'; (c'_1, \text{skip}, S_0) \leftarrow_{p_1} \ldots \leftarrow_{p_k} (c'_1, R_1, V_1); \\
\{ R_2 \mid P' \leftarrow \text{Gen}(\tilde{p}, 1^n); P = P_0 \uplus P'; (c'_2, \text{skip}, S_0) \leftarrow_{p'_1} \ldots \leftarrow_{p'_{k'}} (c'_2, R_2, V_2); 
\end{array}
\]

is negligible in $n$, and

\[
\begin{array}{c}
\{ R_1 \mid P' \leftarrow \text{Gen}(\tilde{p}, 1^n); P = P_0 \uplus P'; (c'_1, R_1, V_1) \leftarrow_{p_1} \ldots \leftarrow_{p_k} (c'_1, R_1, V_1); \\
\{ R_2 \mid P' \leftarrow \text{Gen}(\tilde{p}, 1^n); P = P_0 \uplus P'; (c'_2, R_2, V_2) \leftarrow_{p'_{k'}} \ldots \leftarrow_{p'_{k''}} (c'_2, R_2, V_2); 
\end{array}
\]

Proof. We will prove this lemma in two steps: first by showing that the invariant is preserved across CLIO steps $\sim_p$ and then using that fact, we can show that the invariant is also preserved across CLIO low steps $\sim_P$.

Proof on Step relation $\sim_p$: We will perform induction on the derivation of the steps (which will be finite when used with the low-step rules, i.e., it is well-founded) with the number of steps being $k$ being less than the total number of (possibly high) steps in the context of a single low step.

Our inductive hypothesis will be if $(c_1, \text{skip}, V_1) \equiv_{\ell} (c'_1, \text{skip}, V_2)$ and $\Pi$ is CPA Secure, then,

$$\Pr\left[ (c'_1, R_1, V_1) \not\equiv_{\ell} (c'_2, R_2, V_2) \text{ or } V_1 \not\equiv V_2 \right]$$

is negligible in $n$, and

\[
\begin{array}{c}
\{ R_1 \mid P' \leftarrow \text{Gen}(\tilde{p}, 1^n); P = P_0 \uplus P'; (c'_1, \text{skip}, S_0) \leftarrow_{p_1} \ldots \leftarrow_{p_k} (c'_1, R_1, V_1); \\
\{ R_2 \mid P' \leftarrow \text{Gen}(\tilde{p}, 1^n); P = P_0 \uplus P'; (c'_2, \text{skip}, S_0) \leftarrow_{p'_1} \ldots \leftarrow_{p'_{k'}} (c'_2, R_2, V_2); 
\end{array}
\]

. We must show as our inductive step that if our inductive hypothesis is true, that the following is true:

\[
\begin{array}{c}
\{ R' \mid P' \leftarrow \text{Gen}(\tilde{p}, 1^n); P = P_0 \uplus P'; (c'_1, R_1, V_1) \leftarrow_{p_1} \ldots \leftarrow_{p_k} (c'_1, R_1, V_1); \\
\{ R' \mid P' \leftarrow \text{Gen}(\tilde{p}, 1^n); P = P_0 \uplus P'; (c'_2, R_2, V_2) \leftarrow_{p'_{k'}} \ldots \leftarrow_{p'_{k''}} (c'_2, R_2, V_2); 
\end{array}
\]

The base case $k = 0$ is direct because the initial interactions are a special case of CLIO interactions (i.e., $\text{skip} = \text{skip}$) and $S_0 = S_0$ and we already know by supposition that $c_1 \equiv_{\ell} c_2$.

For the inductive case, we now consider the derivation rule used for the $k + 1$’th step and show that it preserves the inductive hypothesis, assuming it for the $k$’th step.

We note that the single steps may take differing numbers of steps (i.e., $k$ and $k'$). Due to the LOW-TO-HIGH-TO-LOW step rule, though, these differences only occur when $\ell_{\text{out}}$ is high. As a result, the only invariant we need to preserve is confidentiality-only low equivalence between configurations as the high steps do not change the versions, stores, or interactions. We can appeal to the preservation of low equivalence...
of LIO proved by Stefan et al. [54] to conclude the preservation confidentiality-only low equivalence of the standard (i.e., non-store and non-fetch) LIO internal steps. We now consider the low steps that affect the non-standard parts of LIO (i.e., the store and fetch commands).

We also note that since there are only a polynomial number of steps that the resulting sequences of configurations from single steps that do not preserve low equivalent in each step will still together be negligible. As a result, we only need to show that the probability of each step not preserving low equivalence is negligible in n. To that end, we will ignore the traces of steps with negligible probabilities that are not low equivalent.

- **Case Fetch (fetch-exists, fetch-missing, of fetch-replay):**

  In this case, both configurations have a term with an evaluation context hole that is at a fetch command. That is, they are both attempting to fetch an entry from the store with key $v_k$ and $v'_k$. Due to low equivalence they are both fetching the same key, $v_k = v'_k$. As a result, they must each be using one of the following rules: fetch-exists, fetch-missing, or fetch-replay.

  We know by our inductive hypothesis that the distributions of interactions are Clio equivalent. Because the distributions are equal, we can consider steps where the draws are equivalent for values in the erased distributions. We now consider now each case that $v'_k$ transitions with.

  - **Case** fetch-exists and $C(l_1) \subseteq C(f)$: In this case the labeled value will be deserialized with the same way and the same labeled value will be fetched. Since the labeled value is readable by the adversary it must be syntactically equivalent with all but negligible probability, otherwise the interactions would be distinguishable which would be a counter-example to Lemma B.5. As a result, low equivalence will be preserved and both configurations will transition in the same way with all but negligible probability.

  - **Case** fetch-exists and $C(l_1) \not\subseteq C(f)$: In general the value fetched from the store will vary, or it may be the case that only some of the time a value can be deserialized. In these cases, the configurations may transition using this rule and other interactions from the distribution may result in it using another rule. However, if $v'_k$ transitions using another fetch rule (fetch-missing or fetch-replay), the default value will be used. Since secret values can differ and still be low equivalent, the resulting two configurations will still be low equivalent. In each of these cases, no new interactions are produced so the resulting distributions of interactions are still valid Clio interactions by our inductive hypothesis (as they did not change). As a result, low equivalence and the valid interactions invariant is preserved.

  - **Case** fetch-missing or fetch-replay: In these cases, the default labeled value will be used, which by our inductive hypothesis is already low equivalent (due to the configurations being low equivalent). The other configurations will transition in a symmetric way described for the fetch-exists rule.

- **Case** store:

  In this case the distribution of stores and interactions will change so we must show that they remain equivalent. That is, we must show that $(R'_1 | R'_2, R'_1, R'_2)_{\Pi} = (R_1 | R_2, R_1, R_2)_{\Pi}$ where

  \[ R'_1 = \{ \text{put } b_1 : l_1 \text{ at } v_k \cdot R'_1, R_1 \mapsto R_1, \} \]

  \[ (R'_1, b_1 : l_1) \mapsto \text{serialize}_P(\sigma_1, (v_k, v_k, n_1) : l_1) \}

  \[ R'_2 = \{ \text{put } b_2 : l_1 \text{ at } v_k \cdot R'_2, R_2 \mapsto R_2, \} \]

  \[ (R'_2, b_2 : l_1) \mapsto \text{serialize}_P(\sigma_2, (v_k, v_k, n_2) : l_1) \}

  We first note that the entry keys are the same from low equivalence. The versions are equal from Clio low equivalence, i.e., $n_1 = n_2$.

  We also note that the distributions of interactions are valid Clio interactions from our inductive hypothesis for readable labeled values, we can conclude that they are syntactically equivalent values from low equivalence. For non-readable values, the types of the secret values will be the same due to low equivalence (and so the serialized plaintext message will have the same length). As a result the put $b_{b_{0,1}} : l_1$ at $v_k$ will be a valid extension of valid Clio interactions.

  We next consider the creation of category keys (i.e., $R'_1$ and $R'_2$). The initialization of category keys will behave the same way as described for fetching a labeled value: it will either create new keys (if they were corrupted or not there), or skip. It will do this in the same way as the resulting interactions are indistinguishable. For the contents of the category keys, we can divide the parts of the category into deterministic parts (i.e., from Lemma B.3) and secret encryptions.

  With these considerations, we conclude that with all but negligible probability the resulting interactions will be valid Clio interactions.

  Finally, for the versions mappings, we note that they are both updated equivalently (i.e., incremented by the version in the mapping) and the versions mappings originally were equal, so the resulting versions are equal.

  With all cases of the reduction shown to satisfy the proof obligation, we can conclude the inductive hypothesis is true for all steps used in the context of a single low step. We next show the low equivalence invariant on the low step relation.

**Proof on Low-step relation** $\cap_P$: by induction on the number of low steps $j$. Our inductive hypothesis will match our lemma. For all keystores $\mathcal{P}_0$ where $l = \text{authorityOf}(\mathcal{P}_0)$, LIO configurations $c_1, c_2$, strategies $S$, and principals $p$, and $j \in \mathbb{N}$, if $\Pi$ is CPA Secure and $c_1 \equiv_C^{\ell} c_2$, then
interactions. We now must consider the relationship between the probabilities that led to the corresponding low equivalent configurations. For example, it may be the case that a particular distribution of interactions generated from one trace of low steps may be much more likely than another distribution of interactions. However, we can use the preservation of low equivalence to reason about the probabilities of corresponding low equivalent distributions of interactions. We consider the distribution formed from the step function after 1 low step as a running example to make our arguments concrete, shown graphically in the main matter in Section 5.1.

From our inductive hypothesis we know that corresponding low equivalent configurations have indistinguishable distributions of interactions. We now must consider the relationship between the probabilities that led to the corresponding low equivalent configurations.

\[
\Pr\left[\langle c'_1, R_1, V_1 \rangle \neq^C \langle c'_2, R_2, V_2 \rangle \text{ or } V_1 \neq V_2 \right]
\]

is negligible in \(n\), and

\[
\begin{align*}
\{ \bar{R}_1 \} | \mathcal{P}' \leftarrow \text{Gen}(\tilde{\rho}, 1^n); \mathcal{P} = \mathcal{P}_0 \cup \mathcal{P}'\langle c'_1, R_1, V_1 \rangle \leftarrow \text{step}^\ell_\mathcal{P} (c_1, S, j); \\
\{ \bar{R}_2 \} | \mathcal{P}' \leftarrow \text{Gen}(\tilde{\rho}, 1^n); \mathcal{P} = \mathcal{P}_0 \cup \mathcal{P}'\langle c'_2, R_2, V_2 \rangle \leftarrow \text{step}^\ell_\mathcal{P} (c_2, S, j)
\end{align*}
\]

\(n\)

\[\times\]

\[\times\]

is negligible in \(n\), and

\[
\begin{align*}
\{ \bar{R}_1 \} | \mathcal{P}' \leftarrow \text{Gen}(\tilde{\rho}, 1^n); \mathcal{P} = \mathcal{P}_0 \cup \mathcal{P}'\langle c'_1, R_1, V_1 \rangle \leftarrow \text{step}^\ell_\mathcal{P} (c_1, S, 1); \\
\{ \bar{R}_2 \} | \mathcal{P}' \leftarrow \text{Gen}(\tilde{\rho}, 1^n); \mathcal{P} = \mathcal{P}_0 \cup \mathcal{P}'\langle c'_2, R_2, V_2 \rangle \leftarrow \text{step}^\ell_\mathcal{P} (c_2, S, 1)
\end{align*}
\]

\(n\)

\[\times\]

\[\times\]

\[\times\]

\[\times\]

\[\times\]

We must show \(\langle c'_1, R_1, V_1 \rangle \Rightarrow^C \langle c'_2, R_2, V_2 \rangle\) or \(V_1 \neq V_2\). There are two cases we must consider in the low step relation, the Low-To-High-To-Low-Step rule and the Low-To-High-To-Low-Step rule. In the Low-Step, we must show that the inductive hypothesis holds after a single Clio step \(\rightarrow^C\), and for the Low-To-High-To-Low-Step rule must hold for many (finite) Clio steps \(\rightarrow^C\). Note that the Low-Step rule is a special case of the Low-To-High-To-Low-Step rule so we only consider the more general case of preserving the invariant across many steps. To show this, we appeal to the previous proof made to show that the invariant is preserved across Clio steps.

Unlike the single step relation, this includes a strategy interaction on the distribution of stores. Since both interactions receive the indistinguishable distributions of interactions (from Lemma B.5 and the inductive hypothesis) so the resulting distributions from the strategy will also be computationally indistinguishable. That is because if they were not, then the strategy itself could be used as a counter-example for Lemma B.5. In sum, the resulting strategy invocation results in a valid sequence of Clio interactions.

From the previous proof on the single-step relation, we can conclude that \(\langle c'_1, R_1, V_1 \rangle \Rightarrow^C \langle c'_2, R_2, V_2 \rangle\). As a result, we satisfy the inductive hypothesis.

\[\times\]

\[\times\]

\[\times\]

\[\times\]

\[\times\]

With the single low step relation handled we now must consider the distribution of distributions of interactions from the step function. For example, it may be the case that a particular distribution of interactions generated from one trace of low steps may be much more likely than another distribution of interactions. However, we can use the preservation of low equivalence to reason about the probabilities of corresponding low equivalent distributions of interactions. We consider the distribution formed from the step function after 1 low step as a running example to make our arguments concrete, shown graphically in the main matter in Section 5.1.

\]
(e.g., from the diagram \( p_1 \) and \( p_2 \), and also \( p'_1 \) and \( p'_2 \)). If they are similar, then the resulting draws from the distributions will be similar (from low equivalence).

Consider the pairs of low equivalent configurations and the probabilities that led to those configurations (e.g., from the diagram \((c'_1, R_1, V_1)\) with probability \( p_1 \) and \((c'_2, R_2, V_2)\) with probability \( p_2 \)). Consider the ways the configurations can differ probabilistically (e.g., from the diagram, how \( c_1 \) steps to both \( c'_1 \) and \( c''_1 \) and how \( c_2 \) steps to both \( c'_2 \) and \( c''_2 \)). The low step relation is just the probability of the trace of single steps leading to the next low Clio configuration. The Store and Internal-Step rules take steps with probability 1 so they will not cause the low step to differ probabilistically.

Indeed, only the fetching rules Fetch-Exists, Fetch-Missing, and Fetch-Replay rules will cause the configurations to differ probabilistically. In particular they will differ based on the interactions drawn, and as a result differ on how those interactions affect the fetch: if the entry is missing or not deserializable (Fetch-Missing), if the value can be deserialized but the version is old (Fetch-Replay), or if it was successfully deserialized and the version is not old (Fetch-Valid).

Due to our inductive hypothesis we know that the distributions of interactions are valid Clio interactions and as a result are indistinguishable from Lemma B.5. For readable labeled values, the configurations will step with the same probability in lock-step with all but negligible probability, as the readable labeled values will be syntactically equivalent (as the distributions of erased stores are equivalent).

In the case where the label of the labeled value is not readable, the rules used to step may not be the same as they are the results of encrypted values. For example, in one configuration a labeled value may be successfully fetched (using Fetch-Valid) but not in the corresponding configuration (e.g., Fetch-Missing was used). However, as noted above and by our inductive hypothesis, the different rules used will all step to a low equivalent configuration. In addition, though, to the configurations being low equivalent, it is also the case that the sums of the probabilities of all steps taken will be equivalent with all but negligible probability. For example, if \( c_1 \) steps using Fetch-Missing with probability \( p_1 \), and Fetch-Valid with probability \( p_2 \), it is also the case that \( c_2 \) will use the same rules Fetch-Missing with probability \( p_1 \) and Fetch-Valid with probability \( p_2 \) due to indistinguishability of the interactions. That is because if it did not, then an adversary could be constructed to distinguish the interactions based on the proportions of rules used by the Clio semantics. Intuitively, the draws of indistinguishable interactions will produce distributions of indistinguishable steps.

With this reasoning, we conclude that the probabilities of each corresponding single step taking place will be equal (e.g., in the diagram above, \( p_1 = p_2 \) and \( p'_1 = p'_2 \)). So, the resulting distribution of distributions over interactions will be still be valid Clio interactions and so the \( \equiv \) relation holds (and, by Lemma B.5, they are also indistinguishable as a result).
B.3 Indistinguishability Proof

Definition B.7 (Chosen-Term Attack (CTA) Game). Let the random variable $\text{IND}_b(\mathcal{P}, \mathcal{A}, \tilde{p}, j, n)$ denote the output of the following experiment, where $\Pi = (\text{Gen, Enc, Dec, Sign, Verify})$, $\mathcal{A}$ is a non-uniform ppt, $n \in \mathbb{N}$, $b \in \{0, 1\}$:

$$\text{IND}_b(\mathcal{P}_0, \mathcal{A}, \tilde{p}, j, n) =$$

$$\mathcal{P}^* \leftarrow \text{Gen}(\tilde{p}, 1^n); \mathcal{P} = \mathcal{P}_0 \sqcup \mathcal{P}^*;$$

$$t, v_0, v_1, S, A_2 \leftarrow \mathcal{A}(\text{pub}(\mathcal{P})) \text{ such that } v_0 =_\ell v_1$$

and $\triangledown t : \text{Labeled } \tau \rightarrow \text{LIO } \tau'$

and $\triangledown v_0 : \text{Labeled } \tau$

and $\triangledown v_1 : \text{Labeled } \tau$

and $\ell = \text{authorityOf}(\mathcal{P}_0);$

$$\langle c, R_b, V' \rangle \leftarrow \text{step}_t^P(\langle \text{Start}(\mathcal{P}), \text{Clr}(\mathcal{P}) \mid (t v_b), S, j \rangle);$$

$$R_b \leftarrow R_b; \text{ Output } A_2(R_b).$$

We say that $\text{Cl}io$ using $\Pi$ is CTA (Chosen-Term Attack) Secure if for all non-uniform ppt $\mathcal{A}, j \in \mathbb{N}$, keystores $\mathcal{P}$, and principals $\tilde{p}$:

$$\left\{ \text{IND}_0(\mathcal{P}, \mathcal{A}, \tilde{p}, j, n) \right\}_n \approx_n \left\{ \text{IND}_1(\mathcal{P}, \mathcal{A}, \tilde{p}, j, n) \right\}_n.$$

Theorem B.8 (Indistinguishability Theorem). If $\Pi$ is CPA Secure, then $\text{Cl}io$ using $\Pi$ is CTA Secure.

Proof. Direct result of low equivalence (interactions are valid $\text{Cl}io$ interactions, i.e., they satisfy the $\approx$ relation) and Lemma B.5 (indistinguishability of valid $\text{Cl}io$ interactions).
B.4 Leveraged Forgery Lemmas

Lemma B.9 (Starting Label is a Floor). For all keystores \( \mathcal{P}_0 \) and terms \( t \) and strategies \( S \) and principals \( p \) and \( j \),

\[
\Pr\left[ \llbracket (PC(c)) \rrbracket \xrightarrow{I} \llbracket (\text{Start}(\mathcal{P}_0)) \rrbracket \right] \\
\mathcal{P}' \leftarrow \text{Gen}(\bar{p}, 1^n); \\
\mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}''; \\
\langle c, \text{R}, V \rangle \leftarrow \text{step}_{k}^P ((\text{Start}(\mathcal{P}_0), \text{Clr}(\mathcal{P}_0) \mid t), S, j) \right] = 0
\]

Proof. We will prove this lemma in two steps: first by showing that the invariant is preserved across Cl\( \text{i}o \) steps \( \rightsquigarrow_p \) and then using that fact, we can show that the invariant is also preserved across Cl\( \text{i}o \) low steps \( \rightsquigarrow_p \ast \). Our invariant will serve as our inductive hypothesis in both cases.

Proof on Step relation \( \rightsquigarrow_p \ast \): We will perform induction on the derivation of the steps (which will be finite when used with the low-step rules, i.e., it is well-founded) with the number of steps being \( k \) being 1 less than the total number of (possibly high) steps in the context of a single low step, and our inductive hypothesis will be if \( \llbracket (PC(c)) \rrbracket \xrightarrow{I} \llbracket (\text{Start}(\mathcal{P}_0)) \rrbracket \) and,

\[
\Pr\left[ \llbracket (PC(c')) \rrbracket \xrightarrow{I} \llbracket (\text{Start}(\mathcal{P}_0)) \rrbracket \right] \\
\mathcal{P}' \leftarrow \text{Gen}(\bar{p}, 1^n); \\
\mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}''; \\
\langle c, \text{skip}, \Sigma_o \rangle \rightsquigarrow_{p_1} \ldots \rightsquigarrow_{p_k} \langle c', \text{R}', V' \rangle \right] = 0
\]

then,

\[
\Pr\left[ \llbracket (PC(c'')) \rrbracket \xrightarrow{I} \llbracket (\text{Start}(\mathcal{P}_0)) \rrbracket \right] \\
\mathcal{P}' \leftarrow \text{Gen}(\bar{p}, 1^n); \\
\mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}''; \\
\langle c, \text{skip}, \Sigma_o \rangle \rightsquigarrow_{p_1} \ldots \rightsquigarrow_{p_k} \langle c', \text{R}, V \rangle \rightsquigarrow_{p_{k+1}} \langle c'', \text{R}', V' \rangle \right] = 0
\]

The base case \( k = 0 \) is trivial as it is true by supposition. For the inductive case, we now consider the derivation rule used for the \( k + 1 \)th step and show that it preserves the inductive hypothesis, assuming it for the \( k \)th step. We now perform a case analysis on the step used.

- **Case Internal-Step:**
  In this derivation we have that:

\[
\text{INTERNAL STEP} \\
\begin{array}{c}
c' \rightarrow c'' \\
\langle c', \text{R}, V \rangle \rightsquigarrow_{1} \langle c'', \text{R}', V' \rangle \\
\end{array}
\]

By inspection of each of the LIO rules, the label is manipulated in the following ways:
- In \texttt{Unlabel}, the current label is joined with the level of the labeled value, so the flows relation between the current label and the starting label is preserved.
- In \texttt{Reset} the label is returned to its original label. However, from the \texttt{ToLabeled} rule, the label is based on the current label. As a result, since the label is based on a previous step’s current label, and the inductive hypothesis assumes it was true for that point, then the label it is reset to is also satisfies the flow relation to the starting label.
- In all other rules, the current label is not changed, which by supposition satisfies the flow relation.

Proof on Low-Step relation \( \rightsquigarrow_p \ast \) By induction on \( j \).

- **Base Case:** \( j = 1 \): That is, we will prove the following:

\[
\Pr\left[ \llbracket (PC(c)) \rrbracket \xrightarrow{I} \llbracket (\text{Start}(\mathcal{P}_0)) \rrbracket \right] \\
\mathcal{P}' \leftarrow \text{Gen}(\bar{p}, 1^n); \\
\mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}''; \\
\langle c, \text{R}, V \rangle \leftarrow \text{step}_{k}^P ((\text{Start}(\mathcal{P}), \text{Clr}(\mathcal{P}) \mid t), S, 1) \right] = 0
\]

There are two cases we must consider in the low step relation, the \texttt{LowStep} rule and the \texttt{Low-to-High-to-Low-Step} rule. In the \texttt{Low-Step}, we must show that the inductive hypothesis holds after a single Cl\( \text{i}o \) step \( \rightsquigarrow_p \), and for the \texttt{Low-to-High-to-Low-Step} rule must hold for many (finite) Cl\( \text{i}o \) steps \( \rightsquigarrow_p \ast \). Note that the \texttt{LowStep} rule is a special case of the \texttt{Low-to-High-to-Low-Step} rule so we only consider the more general case of preserving the invariant across many steps. To show this, we appeal to the previous proof made to show that the invariant is preserved across Cl\( \text{i}o \) steps. As a result, we have that:

\[
\Pr\left[ \llbracket (PC(c')) \rrbracket \xrightarrow{I} \llbracket (\text{Start}(\mathcal{P}_0)) \rrbracket \right] \\
\mathcal{P}' \leftarrow \text{Gen}(\bar{p}, 1^n);
\]

By inspection of each of the LIO rules, the label is manipulated in the following ways:
- In \texttt{Unlabel}, the current label is joined with the level of the labeled value, so the flows relation between the current label and the starting label is preserved.
- In \texttt{Reset} the label is returned to its original label. However, from the \texttt{ToLabeled} rule, the label is based on the current label. As a result, since the label is based on a previous step’s current label, and the inductive hypothesis assumes it was true for that point, then the label it is reset to is also satisfies the flow relation to the starting label.
- In all other rules, the current label is not changed, which by supposition satisfies the flow relation.
\( P = P_0 \cup P'; \langle c, \text{skip}, \Sigma_0 \rangle \leadsto_{p_1} \ldots \leadsto_{p_k} \langle c', R, V \rangle \) = 0

From the previous proof on the single-step relation, we can conclude that \( I(\text{PC}(c)) \not \sqsubseteq I(\text{Start}(P_0)) \). As a result, we satisfy the inductive hypothesis.

- **Inductive Case:** \( j = k+1 \): That is, we will prove the following:

\[
\Pr[I(\text{PC}(c)) \not \sqsubseteq I(\text{Start}(P_0)) \mid P' \leftarrow \text{Gen}(\hat{\rho}, 1^n); P = P_0 \cup P'; \langle c, R, V \rangle \leftarrow \text{step}_{\ell}((\text{Start}(P), \text{Clr}(P) \mid t), S, k + 1)] = 0
\]

We can expand the step metafunction to be

\[
\langle c'', R'', V' \rangle, p : p'
\]

where

\[
\langle \langle c', R, V \rangle, p \rangle \in \text{step}_{\ell}(c, S, k); \\
R' = S(R); \\
\langle \langle c', R, V \rangle, R' \rangle \leadsto_{p'} \langle c'', R'', V' \rangle
\]

The strategy on the stores does not affect the current label. After this adversary interaction, we can invoke the single-step lemma result here to conclude that \( I(\text{PC}(c'')) \not \sqsubseteq I(\text{Start}(P_0)) \). As a result, the inductive hypothesis is true.

With all cases accounted for in the low step relation, and the single-step relation, we can conclude the proof.

\( \square \)
B.5 Leveraged Forgery Security Proof

**Definition B.10 (Values function).** Define the Values function as follows:

\[
\text{Values}(\text{put} \cdot b \cdot l_1 \cdot v \cdot k \cdot R) = \begin{cases} 
\text{put} \cdot b \cdot l_1 \cdot v \cdot k \cdot \text{Values}(R) & \text{if } \langle v, \text{deserialize}(R), \tau \rangle = \text{deserialize} \cdot P \cdot R \\
\text{Values}(\text{put} \cdot b \cdot l_1 \cdot v \cdot k \cdot R) & \text{otherwise}
\end{cases}
\]

**Theorem B.11 (Existential forgery under chosen message attack).** For all keystores \( P_0 \), principals \( p \) in principal sets \( \hat{p} \), and \( j \) if \( \Pi \) is secure against existential forgery under chosen message attacks, then for all \( \text{I}(l_2) \subseteq \text{I}(l_1) \subseteq \text{I}(\text{authorityOf}(P_0)) \) and \( p \),

\[
\Pr \left[ b \cdot l_1 \cdot \in \text{Values}_P(\bar{R}) \text{ and } b \cdot l_2 \cdot \notin \text{Values}_P(\bar{R}) \right] \]

\[
P' \leftarrow \text{Gen}(\{p\}, 1^n); \quad P = P_0 \sqcup P';
\]

\[
t, S, A_2 \leftarrow A(\text{pub}(P));
\]

\[
\langle c, R, V \rangle \leftarrow \text{step}_{P'}(\langle \text{Start}(P), \text{Clr}(P) \mid t, S, j \rangle);
\]

\[
\bar{R} \leftarrow R;
\]

\[
t', S' \leftarrow A_2(\bar{R});
\]

\[
\langle c', R', V' \rangle \leftarrow \text{step}_{P}(\langle \text{Start}(P_0), \text{Clr}(P) \mid t', S', j \rangle);
\]

\[
\bar{R}' \leftarrow R'
\]

is negligible in \( n \).

**Proof.** We consider the level required to produce a valid signature during a store operation. The signature must be valid for a \( p \in \hat{p} \). By inspection of the C\textsc{lio} semantics, the only way to a valid signature would occur in the interaction is during the store operation, which uses the labeled value’s label, or by the strategy.

According to the store operation, current label must be bounded above by the label of the labeled value (i.e., \( l_\text{cur} \subseteq l_1 \)). For integrity, this means the current label’s integrity component \( I \) must be at least as trustworthy as the principal \( p \).

By the previous lemma, we can conclude that the current integrity label will never be at a level \( I' \) such that \( I' \sqsubseteq I(\text{Start}(P_0)) \)

This means that the level of the C\textsc{lio} computation would need to be at least \( I = I(\text{authorityOf}(P_0) \sqcup \{p \mapsto \text{Gen}(1^n)\}) \)

By unpacking the definition of \( \text{Start}(P \sqcup \{p \mapsto \text{Gen}(1^n)\}) \), this integrity label satisfies the following relation

\[
I \sqsubseteq I(\text{Start}(P_0))
\]

Which we have shown is impossible to reach. As a result, the current label’s integrity level will never be at a level where it can sign the value using \( p \)’s private signing key.

As a result, the only way a high integrity value could be in the challenge store \( \sigma' \) and not in the original store \( \sigma \) would require the strategy to forge a signature itself without C\textsc{lio}’s assistance. Since this occurs with only negligible probability, we have satisfied the proof obligation. □
C Implementation Details

In this section we discuss additional implementation details of our prototype.

C.1 Storing and Fetching

For each category used in the program we generate a symmetric key and two RSA key pairs: an encryption/decryption key pair and a signing/verification key pair. This information is stored in the database after being asymmetrically encrypted and then signed as described in Section 4.1. Category key generation relies on the RSA key pairs for each principal involved, which should be supplied by the user in the form of an initial keystore when the Clio computation starts.

After the relevant IFC effects have been performed, storing a labeled value involves fetching the symmetric key for each category in its confidentiality clause as well as the signature keys that correspond to each category in its integrity clause, potentially generating these on the fly. The labeled value is serialized to a bitstring, then RSA-PSS signed by at least one principal per integrity category, and finally AES256-CTR onion-encrypted using the symmetric key for each confidentiality category. Fetching involves the dual operations, i.e., symmetric decryption and RSA-PSS signature verification.

In order to avoid problems with improperly escaped strings, we encode every bitstring in base64.

C.2 User API

Our library provides all the Clio operations described in the paper, plus a few extra functions that are necessary to glue Clio code with the rest of the program. Here are some of the most important ones.

Clio code can be run using the evalCLIO function. This function takes two arguments: a record initialState of type CLIOState and a Clio computation m. The record initialState provides initial values for the current label, the current clearance, the keystore, the version map and the store label. The function simply establishes a connection with a Redis server and executes m using that database as the store and initialState as the local state.

In order to generate keystores, we provide the utility function initializeKeyMapIO. This function takes a list of principals as argument, and produces a keystore with fresh asymmetric key pairs for all of them. Our prototype does not provide means to store these keystores beyond the execution of the program, but it would be straightforward for users to implement this functionality in their own programs, or with their own PKI.