Cryptographically Secure Information Flow Control on Key-Value Stores

Lucas Waye
Harvard University
Cambridge, Massachusetts
lwaye@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., OpenSSL[1]) that are very similar to the interfaces for establishing unencrypted connections. These 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 untrustworthy 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

*Work done while author was at Harvard University.
[1]https://www.openssl.org/

CCS '17, October 30-November 3, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-4946-8/17/10.
https://doi.org/10.1145/3133956.3134036

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 `fetch`s and `store`s to a Clio store.

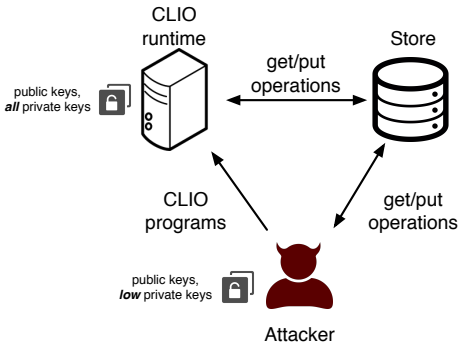

Figure 1: Clio threat model. Attackers write Clio programs, read from and write to the store, and observe the runtime's interactions.

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

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 $(\gg\!=)$ (read as "bind"). The expression return x denotes a computation that returns the value denoted by x, performing no side-effects. The function $(\gg\!=)$ is used to *sequence* computations. Specifically, $t \gg\!= \lambda x.t'$ takes the result produced by t and applies function $\lambda x.t'$ 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 IFC-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 ($\sqsubseteq$) set with least upper bounds ($\sqcup$) and greatest lower bounds ($\sqcap$). A runtime monitor maintains as part of its state a distinguished label l_{cur} 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 l_{cur} observes an entity with label l, its current label is increased (if necessary) to the least upper bound of the two labels, written $l_{\text{cur}} \sqcup l$. 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.

$$
\begin{array}{rrl}
\text{Ground Value:} & \underline{v} & ::= \mathsf{true} \mid \mathsf{false} \mid () \mid l \mid (\underline{v}, \underline{v}) \\
\text{Value:} & v & ::= \underline{v} \mid (v, v) \mid x \mid \lambda x.t \mid t^{\mathsf{CLIO}} \mid \langle v : l \rangle \\
\text{Term:} & t & ::= v \mid (t, t) \mid t\ t \mid \mathsf{fix}\ t \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \\
 & & \quad \mid t_1 \sqcup t_2 \mid t_1 \sqcap t_2 \mid t_1 \sqsubseteq t_2 \\
 & & \quad \mid \mathsf{return}\ t \mid t \gg\!= t \\
 & & \quad \mid \mathsf{label}\ t\ t \mid \mathsf{labelOf}\ t \mid \mathsf{unlabel}\ t \\
 & & \quad \mid \mathsf{getLabel} \mid \mathsf{getClearance} \mid \mathsf{lowerClearance}\ t \\
 & & \quad \mid \mathsf{toLabeled}\ t\ t \mid {}^{l}_{l}\{^{l}\ t\ \} \\
 & & \quad \mid \mathsf{store}\ t\ t \mid \mathsf{fetch}_{\tau}\ t\ t \\
\text{Ground Type:} & \underline{\tau} & ::= \mathsf{Bool} \mid () \mid \mathsf{Label} \mid (\underline{\tau}, \underline{\tau}) \\
\text{Type:} & \tau & ::= \underline{\tau} \mid (\tau, \tau) \mid \tau \to \tau \mid \mathsf{CLIO}\ \tau \mid \mathsf{Labeled}\ \tau
\end{array}
$$

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 ($l_{\text{cur}} \sqsubseteq l$).

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 l m will run m to completion, and produce a *labeled value* $\langle v : l \rangle$, 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 l m. In general, given a labeled value $\langle v : l \rangle$, 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 $\mathcal{A}$ that can observe inputs and outputs with confidentiality label at most $l_{\mathcal{A}}$ learns nothing about any input to the program with label l such that $l \not\sqsubseteq l_{\mathcal{A}}$. 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 τ. Computation on

LabelOf

$$\frac{}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{labelOf}\ (\langle t : l_1 \rangle)\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid l\rangle}$$

Return

$$\frac{}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{return}\ t\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid t\ ^{\mathsf{CLIO}}\rangle}$$

Bind

$$\frac{}{\langle l_{\text{cur}}, l_{\text{clr}} \mid t_1\ ^{\mathsf{CLIO}} \gg\!= t_2\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid (t_2\ t_1)\rangle}$$

Label

$$\frac{l_{\text{cur}} \sqsubseteq l_1 \qquad l_1 \sqsubseteq l_{\text{clr}}}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{label}\ l_1\ \underline{v}\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{return}\ (\langle \underline{v} : l_1 \rangle)\rangle}$$

Unlabel

$$\frac{l_{\text{cur}} \sqcup l_1 = l_2 \qquad l_2 \sqsubseteq l_{\text{clr}}}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{unlabel}\ (\langle t_2 : l_1 \rangle)\rangle \longrightarrow \langle l_2, l_{\text{clr}} \mid \mathsf{return}\ t_2\rangle}$$

ToLabeled

$$\frac{l_{\text{cur}} \sqsubseteq l_1 \qquad l_1 \sqsubseteq l_{\text{clr}}}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{toLabeled}\ l_1\ t\rangle \longrightarrow \langle l_{\text{cur}}, l_{\text{clr}} \mid {}^{l_{\text{cur}}}_{l_{\text{clr}}}\{^{l_1}\ t\ \}\rangle}$$

Reset

$$\frac{l_{\text{cur}} \sqsubseteq l_2}{\langle l_{\text{cur}}, l_{\text{clr}} \mid {}^{l_1}_{l_3}\{^{l_2}\ t\ ^{\mathsf{CLIO}}\ \}\rangle \longrightarrow \langle l_1, l_3 \mid \mathsf{label}\ l_2\ t\rangle}$$

Figure 3: Clio language semantics (selected rules).

labeled values occur in the CLIO monad using the return and ($\gg\!=$) monadic operators. The nonterminals t^{CLIO} and ${}^{l}_{l'}\{^{l''}\ t\ \}$ 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*, $\underline{v}$. 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 $\vdash t : \tau$ since they are mostly standard[2]. 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 $\longrightarrow$ over Clio configurations (Figure 3)[3]. Configurations are of the form $\langle l_{\text{cur}}, l_{\text{clr}} \mid t\rangle$, where l_{cur} is the current label and t is the Clio term being evaluated. Label l_{clr} is the *current clearance* and is an upper bound on the current label l_{cur}. The clearance allows a programmer to specify an upper bound for information that a computation is allowed to access. We write $c \longrightarrow c'$ to express that configuration c can take a reduction step to configuration c'. We define $\longrightarrow^*$ as the reflexive and transitive closure of $\longrightarrow$. Given configuration $c = \langle l_{\text{cur}}, l_{\text{clr}} \mid t\rangle$ we write $\text{PC}(c)$ for l_{cur}, the current label of c.

Rules return and bind encode the core monadic operations. The intermediate value t^{CLIO} is used to represent a Clio computation which produces the term t, without any further effects on the configuration. In rule label, the operation label l $\underline{v}$ returns a labeled value with label l holding $\underline{v}$ ($\langle \underline{v} : l \rangle$), provided that the current label flows to l ($l_{\text{cur}} \sqsubseteq l$) and l flows to the current clearance ($l \sqsubseteq l_{\text{clr}}$). Note that we force the second argument to be a ground value, i.e. it should be fully normalized. Rule unlabel expresses that, given a labeled value lv with label l, the operation unlabel lv returns the value stored in lv and updates the current label to $l_{\text{cur}} \sqcup l$, to capture the fact that a value with label l has been read, provided that this new label flows to the current clearance ($l \sqsubseteq l_{\text{clr}}$). 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 ($l_{\text{cur}} \sqsubseteq l_1 \sqsubseteq l_{\text{clr}}$) and then wrap t in a compartment using the special syntactic form ${}^{l_{\text{cur}}}_{l_{\text{clr}}}\{^{l_1}\ t\ \}$, 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 $t_1\ ^{\mathsf{CLIO}}$. Next, the rule reset evaluates the term ${}^{l_{\text{cur}}}_{l_{\text{clr}}}\{^{l_1}\ t_1\ ^{\mathsf{CLIO}}\ \}$, first checking that the current label flows to the target of the current toLabeled ($l_{\text{cur}} \sqsubseteq l_2$). 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 $\langle l_c, l_i, l_a\rangle$ represents a policy with confidentiality l_c, integrity l_i, and availability l_a. Information labeled with $\langle l_c, l_i, l_a\rangle$ can be read by l_c, is vouched for by l_i, and is hosted by l_a. We write $\mathbb{C}(l)$, $\mathbb{I}(l)$, $\mathbb{A}(l)$ for the confidentiality, integrity, and availability components of l, respectively. Each component is a conjunction of disjunctions of principal names, i.e., a formula in conjunctive normal form. A disjunction $A \vee B$ in the confidentiality component means that either A or B can read the data; in the integrity component, it means that one of A or B vouch for the data, but none of them take sole responsibility; in terms of availability, it means that one of A or B can deny access to the data. Conjunctions $A \wedge B$ mean that only A and B together can read the data (confidentiality), that they jointly vouch for the data (integrity), or that they can jointly deny access to the data (availability). Data may flow between differently labeled entities, but only those with more *restrictive* policies: those readable, vouched for, or hosted by fewer entities. A label $\langle l_c, l_i, l_a\rangle$ can flow to any label where the confidentiality component is at least as sensitive than l_c, the integrity component is no more trustworthy than l_i, and the availability is no more than l_a, i.e. $l_1 \sqsubseteq l_2$ if and only if $\mathbb{C}(l_2) \implies \mathbb{C}(l_1)$, $\mathbb{I}(l_1) \implies \mathbb{I}(l_2)$, and $\mathbb{A}(l_1) \implies \mathbb{A}(l_2)$. We use logical implication because it matches the intuitive meaning of disjunctions and conjunctions, e.g., data readable by $A \vee B$ is less confidential than data readable only by A, and data vouched for by $A \wedge B$ is more trustworthy than data vouched for only by A. In the rest of the paper, we consider only Clio computations that work on labels of this form.

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 and 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 τ.

[2]Complete definitions given in the technical report [59].
[3]The rest can be found in the technical report [59].

STORE

$$\frac{l_{\text{cur}} \sqsubseteq \ell \qquad l_{\text{cur}} \sqsubseteq l_1 \qquad \alpha = \mathsf{put}\ \langle \underline{v} : l_1 \rangle\ \mathsf{at}\ \underline{v}_k}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{store}\ \underline{v}_k\ \langle \underline{v} : l_1 \rangle \rangle \xrightarrow{\alpha} \langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{return}\ () \rangle}$$

FETCH-VALID

$$\frac{\mathbb{A}(\ell) \sqsubseteq^A \mathbb{A}(l_d) \qquad \alpha = \mathsf{got}_{\,\tau}\ \langle \underline{v} : l \rangle\ \mathsf{at}\ \underline{v}_k \qquad l \sqsubseteq l_d}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{fetch}_{\,\tau}\ \underline{v}_k\ \langle \underline{v}_d : l_d \rangle \rangle \xrightarrow{\alpha} \langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{return}\ \langle \underline{v} : l_d \rangle \rangle}$$

FETCH-INVALID

$$\frac{\mathbb{A}(\ell) \sqsubseteq^A \mathbb{A}(l_d) \qquad (\alpha = \mathsf{nothing\text{-}at}\ \underline{v}_k) \text{ or } (\alpha = \mathsf{got}_{\,\tau}\ \langle \underline{v} : l \rangle\ \mathsf{at}\ \underline{v}_k \text{ and } l \not\sqsubseteq l_d)}{\langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{fetch}_{\,\tau}\ \underline{v}_k\ \langle \underline{v}_d : l_d \rangle \rangle \xrightarrow{\alpha} \langle l_{\text{cur}}, l_{\text{clr}} \mid \mathsf{return}\ \langle \underline{v}_d : l_d \rangle \rangle}$$

Figure 4: CLIO language semantics (store and fetch rules).

Semantics for fetch and store are shown in Figure 4. We modify the semantics to be a labeled transition system, where step relation $\xrightarrow{\alpha}$ is annotated with *store events* α. Store event α is one of:

- skip (representing no interaction with the store, i.e., an internal step; we typically elide skip for clarity),
- put $\langle \underline{v} : l \rangle$ at $\underline{v}_k$ (representing putting a labeled ground value $\langle \underline{v} : l \rangle$ indexed by $\underline{v}_k$),
- got$_\tau$ $\langle \underline{v} : l \rangle$ at $\underline{v}_k$ (representing reading a labeled value from the store indexed by $\underline{v}_k$), or
- nothing-at $\underline{v}_k$ representing no value is indexed by $\underline{v}_k$.

Labeling transitions with store events allows us to cleanly factor out the implementation of the store, enabling us to easily use either an idealized (non-cryptographic) store, or a store that uses cryptography to help enforce security guarantees. We describe the semantics of store events in both these settings later.

We associate a label ℓ with the store. Intuitively, store level ℓ describes how trusted the store is: it represents the inherent protections provided by the store and the inherent trust by the store in CLIO. For example, the store may be behind an organization's firewall so data is accessible only to organization members due to an external access control mechanism (i.e., the firewall), so CLIO can safely store the organization's information there. Dually, there may be integrity requirements that CLIO is trusted to uphold when writing to the store. For example, the store may be used as part of a larger system that uses the store to perform important operations (e.g., ship customer orders). Thus the integrity component of the store label is a bound on the untrustworthiness of information that CLIO should write to the store (e.g., CLIO should not put unendorsed shipping requests in the store). The availability component of the store label specifies a bound on who is able to corrupt information in the store and thus make it unavailable. (Note that we are concerned with *information availability* rather than *system availability*.) In general, this would describe all the principals who have direct and indirect write-access to the store.

Rule STORE (Figure 3) is used to put a labeled value $\langle \underline{v} : l \rangle$ in the store, indexed by key $\underline{v}_k$. We require that the current label l_{cur} is bounded above by store level ℓ. In terms of confidentiality, this means that any information that may be revealed by performing the store operation (i.e., l_{cur}) is permitted to be learned by users of the store. For integrity, the decision to place this value in the 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., $l_{\text{cur}} \sqsubseteq l_1$). 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_{cur} is an upper bound on the information that led to the decision to perform the store, and l_1 bounds who may learn the labeled value.

For command fetch$_\tau$ $\underline{v}_k$ $\langle \underline{v}_d : l_d \rangle$, labeled value $\langle \underline{v}_d : l_d \rangle$ 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 $\langle \underline{v}_d : l_d \rangle$ (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 $\langle \underline{v} : l \rangle$ where $l \not\sqsubseteq l_d$, then the fetch command evaluates to $\langle \underline{v}_d : l_d \rangle$ instead. This allows programmers to specify bounds on information they are willing to read from the store.

Rule FETCH-VALID is used when a labeled value is successfully fetched from the store. Store event got$_\tau$ $\langle \underline{v} : l \rangle$ at $\underline{v}_k$ indicates that the store was able to return labeled value $\langle \underline{v} : l \rangle$ indexed by the key $\underline{v}_k$. Rule 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., $\mathbb{A}(\ell) \sqsubseteq^A \mathbb{A}(l_d)$) 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 CLIO program.[4] 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 $\langle c, \sigma \rangle \rightsquigarrow \langle c', \sigma' \rangle$ where $\langle c, \sigma \rangle$ and $\langle c', \sigma' \rangle$ are pairs of a CLIO configuration c and an ideal store σ. An ideal store σ maps ground values $\underline{v}_k$ to labeled ground values $\langle \underline{v} : l \rangle$. If a store doesn't contain a mapping for an index $\underline{v}_k$, we represent that as mapping it to the distinguished value $\perp$.

Store events are used to communicate with the store. When a put $\langle \underline{v} : l \rangle$ at $\underline{v}_k$ event is emitted, the store is updated appropriately. When the CLIO computation issues a fetch command, the store provides the appropriate event (i.e., either provides event nothing-at $\underline{v}_k$ or event got$_\tau$ $\langle \underline{v} : l \rangle$ at $\underline{v}_k$ for an appropriate labeled value $\langle \underline{v} : l \rangle$). For CLIO computation steps that do not interact with the store, store event skip is emitted, and the store is not updated.

[4] Complete formal definitions in the technical report [59].

$$\textsc{Low-Step}\quad \frac{\langle c, \bar{I}(\sigma)\rangle \rightsquigarrow \langle c', \sigma'\rangle \quad \mathrm{PC}(c) \sqsubseteq \ell \quad \mathrm{PC}(c') \sqsubseteq \ell}{(\langle c, \sigma\rangle, \bar{I}) \curvearrowright \langle c', \sigma'\rangle}$$

$$\textsc{Low-to-High-to-Low-Step}\quad \frac{\langle c, \bar{I}(\sigma)\rangle \rightsquigarrow \langle c_0, \sigma_0\rangle \quad \langle c_0, \sigma_0\rangle \rightsquigarrow \ldots \rightsquigarrow \langle c_j, \sigma_j\rangle \quad \forall_{0 \le i < j}.\ \mathrm{PC}(c_i) \not\sqsubseteq \ell \quad \mathrm{PC}(c_j) \sqsubseteq \ell}{(\langle c, \sigma\rangle, \bar{I}) \curvearrowright \langle c_j, \sigma_j\rangle}$$

$$\bar{I} ::= I \cdot \bar{I} \mid I$$

$$I ::= \mathsf{skip} = \lambda\sigma.\sigma$$

$$\mid \mathsf{put}\ \langle \underline{v}{:}l_1\rangle\ \mathsf{at}\ \underline{v}' = \lambda\sigma.\ \sigma[\underline{v}' \mapsto \langle \underline{v}{:}l_1\rangle] \text{ s.t. } \mathbb{I}(\ell) \sqsubseteq^I \mathbb{I}(l_1)$$

$$\mid \mathsf{corrupt}\ \underline{v}_1, \ldots \underline{v}_n = \lambda\sigma.\sigma[\underline{v}_1 \mapsto \bot; \ldots\ \underline{v}_n \mapsto \bot]$$

Figure 5: Adversary Interactions and Low Steps

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 ℓ 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 $\langle \underline{v}{:}l_1\rangle$ in the store where the confidentiality level of label l_1 is at least as confidential as the store level ℓ (i.e., $\mathbb{C}(l_1) \sqsubseteq^C \mathbb{C}(\ell)$).
- An adversary can put labeled values $\langle \underline{v} : l_1\rangle$ in the store (with arbitrary ground value index $\underline{v}_k$) provided the integrity level of store level ℓ is at least as trustworthy as the integrity of label l_1 (i.e., $\mathbb{I}(\ell) \sqsubseteq^I \mathbb{I}(l_1)$).

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 $\langle \underline{v}{:}l_1\rangle$ at $\underline{v}'$ and corrupt $\underline{v}_1, \ldots \underline{v}_n$, which, respectively: do nothing; put a labeled value in the store; and delete the mappings for entries at indices $\underline{v}_1$ to $\underline{v}_n$. 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 $\curvearrowright$ 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 ℓ. (By contrast, a *high step* is when the current label can not flow to ℓ.) This captures the threat model assumption that the attacker cannot observe timing. Rules Low-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 $\mathcal{P} : p \mapsto (\{0,1\}^*, \{0,1\}^*_\bot)$, 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 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 $\mathsf{authorityOf}(\mathcal{P})$ returns a label where each component (confidentiality, integrity, and availability) is the conjunction of all principals for which keystore $\mathcal{P}$ has the private key. We also use the keystore to determine the starting label of a Clio program $\mathsf{Start}(\mathcal{P})$ and the least restrictive clearance $\mathsf{Clr}(\mathcal{P})$, 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* $C_1 \wedge ... \wedge C_n$. Categories are disjunctions of principals $p_1 \vee ... \vee p_n$. 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 $C_1 \wedge ... \wedge C_n$ 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 $C_2 \wedge ... \wedge C_n$. 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 $\langle\langle l_c, l_i, l_a\rangle : \underline{v}\rangle$, the value $\underline{v}$ is signed according to formula l_i. 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 $\rightsquigarrow_p$. 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 $\langle c, \mathbb{R}, \mathbf{V}\rangle$ of a Clio configuration c, a *distribution* of *sequences* of *real interactions* with the store $\mathbb{R}$, and a version map $\mathbf{V}$. 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 $\overline{R}$) 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 $\mathsf{put}\ ck\ \mathsf{at}\ C$ represents storing of a category key. These interactions arise from the serialize metafunction, which may create new category keys. Note that the interaction $\mathsf{put}\ \langle b : l\rangle\ \mathsf{at}\ \underline{v}_k$ 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

$$\{\!|\ f(X_1, ..., X_n) \mid X_1 \leftarrow D_1;\ ...\ X_n \leftarrow D_n\ |\!\}$$

[6] Defined formally in the technical report [59].

[7] The Clio runtime ensures that the first time a category key for a given category is required, it will be because data confidential to the category or vouched for by the category is being written to the store, and thus the computation has access to at least one category member's private key. Note that any computation with the authority of a category member has the authority of the category.

[8] "Encrypt-then-sign"issues (e.g., [1]) do not apply here as the threat model (i.e., signed encrypted messages implying authorship) is different.

Internal-Step

$$\frac{c \longrightarrow c'}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_1 \langle c', \mathbb{R}, \mathbf{V}\rangle}$$

Store

$$\frac{\begin{array}{c} c \xrightarrow{\mathsf{put}\ \langle \underline{v} : l_1 \rangle\ \mathsf{at}\ \underline{v}_k} c' \\ n = \mathrm{increment}(\mathbf{V}(\underline{v}_k)) \qquad \mathbf{V}' = \mathbf{V}[\underline{v}_k \mapsto n] \\ \mathbb{R}' = \{\!\!\{ \mathsf{put}\ \langle b : l_1 \rangle\ \mathsf{at}\ \underline{v}_k \cdot \overline{R'} \cdot \overline{R} \;\big|\; \overline{R} \leftarrow \mathbb{R}; \\ (\overline{R'}, \langle b : l_1 \rangle) \leftarrow \mathsf{serialize}_{\mathcal{P}}(\sigma, \langle (\underline{v}, \underline{v}_k, n) : l_1 \rangle) \}\!\!\} \end{array}}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_1 \langle c', \mathbb{R}', \mathbf{V}'\rangle}$$

Fetch-Exists

$$\frac{\begin{array}{c} c \xrightarrow{\mathsf{got}_\tau\ \langle \underline{v} : l_1 \rangle\ \mathsf{at}\ \underline{v}_k} c' \qquad n \not< \mathbf{V}(\underline{v}_k) \\ (\sigma, p) \in \{\!\!\{ \overline{R}(\emptyset) \mid \overline{R} \leftarrow \mathbb{R} \}\!\!\} \qquad p > 0 \\ \langle (\underline{v}, \underline{v}_k, n) : l_1 \rangle = \mathsf{deserialize}_{\mathcal{P}}(\sigma, \sigma(\underline{v}_k), \tau) \end{array}}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_p \langle c, \mathbb{R}, \mathbf{V}\rangle}$$

Fetch-Missing

$$\frac{\begin{array}{c} c \xrightarrow{\mathsf{nothing\text{-}at}\ \underline{v}_k} c' \\ (\sigma, p) \in \{\!\!\{ \overline{R}(\emptyset) \mid \overline{R} \leftarrow \mathbb{R} \}\!\!\} \qquad p > 0 \\ \mathsf{deserialize}_{\mathcal{P}}(\sigma, \sigma(\underline{v}_k), \tau)\ \text{undefined} \end{array}}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_p \langle c, \mathbb{R}, \mathbf{V}\rangle}$$

Fetch-Replay

$$\frac{\begin{array}{c} c \xrightarrow{\mathsf{nothing\text{-}at}\ \underline{v}_k} c' \qquad \underline{v}'_k \neq \underline{v}_k\ \text{or}\ n < \mathbf{V}(\underline{v}_k) \\ (\sigma, p) \in \{\!\!\{ \overline{R}(\emptyset) \mid \overline{R} \leftarrow \mathbb{R} \}\!\!\} \qquad p > 0 \\ \langle (\underline{v}, \underline{v}'_k, n) : l_1 \rangle = \mathsf{deserialize}_{\mathcal{P}}(\sigma, \sigma(\underline{v}_k), \tau) \end{array}}{\langle c, \mathbb{R}, \mathbf{V}\rangle \rightsquigarrow_p \langle c, \mathbb{R}, \mathbf{V}\rangle}$$

Figure 6: Real Clio Semantics

to describe the probability distribution over the function f with inputs of random variables $X_1, ..., X_n$ 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 $\mathbb{R}'$ 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,

$$(\sigma, p) \in \{\!\!\{ \overline{R}(\emptyset) \mid \overline{R} \leftarrow \mathbb{R} \}\!\!\}$$

in each of these rules means that store σ has probability p of being produced (by drawing interaction sequence $\overline{R}$ from distribution $\mathbb{R}$ and applying $\overline{R}$ to the empty store $\emptyset$ to give store σ).

Which rule is used for a fetch operation depends on the state of the store, and so the transitions may have probability less than

Low-Step

$$\frac{\begin{array}{c} \mathbb{R}' = \{\!\!\{ \overline{R}_A \cdot \overline{R} \mid \overline{R}_A \leftarrow \mathbb{R}_A;\ \overline{R} \leftarrow \mathbb{R} \}\!\!\} \\ \langle c, \mathbb{R}', \mathbf{V}\rangle \rightsquigarrow_p \langle c', \mathbb{R}'', \mathbf{V}'\rangle \\ \mathrm{PC}(c) \sqsubseteq \mathbb{C}(\ell) \qquad \mathrm{PC}(c') \sqsubseteq \mathbb{C}(\ell) \end{array}}{(\langle c, \mathbb{R}, \mathbf{V}\rangle, \mathbb{R}_A) \curvearrowright_p \langle c', \mathbb{R}'', \mathbf{V}'\rangle}$$

Low-to-High-to-Low-Step

$$\frac{\begin{array}{c} \mathbb{R}' = \{\!\!\{ \overline{R}_A \cdot \overline{R} \mid \overline{R}_A \leftarrow \mathbb{R}_A;\ \overline{R} \leftarrow \mathbb{R} \}\!\!\} \\ \langle c, \mathbb{R}', \mathbf{V}\rangle \rightsquigarrow_{p_0} \langle c_0, \mathbb{R}_0, \mathbf{V}\rangle) \\ \langle c_0, \mathbb{R}_0, \mathbf{V}_0\rangle \rightsquigarrow_{p_1} \ldots \rightsquigarrow_{p_j} \langle c_j, \mathbb{R}_j, \mathbf{V}_j\rangle \\ \forall_{0 \leq i < j}.\ \mathrm{PC}(c_i) \not\sqsubseteq \ell \qquad \mathrm{PC}(c_j) \sqsubseteq \ell \qquad p = \Pi_{0 \leq i \leq j}\, p_i \end{array}}{(\langle c, \mathbb{R}, \mathbf{V}\rangle, \mathbb{R}_A) \curvearrowright_p \langle c_j, \mathbb{R}_j, \mathbf{V}_j\rangle}$$

$$\begin{array}{ll} \text{Interactions:} & R ::= \mathsf{skip} = \lambda\sigma.\ \sigma \\ & \mid \mathsf{put}\ ck\ \mathsf{at}\ C = \lambda\sigma.\ \sigma[C \mapsto ck] \\ & \mid \mathsf{put}\ \langle b : l \rangle\ \mathsf{at}\ \underline{v}_k = \lambda\sigma.\ \sigma[\underline{v}_k \mapsto \langle b : l \rangle] \\ \text{Strategies:} & \mathcal{S}\ : \mathbb{R} \to \mathbb{R} \end{array}$$

$$\begin{array}{l} \mathsf{step}^{\mathcal{P}}_{\ell}(c_0, \mathcal{S}, 1) \quad = \Big\{ \big(\langle c_1, \mathbb{R}_1, \mathbf{V}_1\rangle, p_0 \cdot p_1\big) \;\Big|\; \\ \quad (\langle c_0, \{(\mathsf{skip}, 1)\}, \Sigma_0\rangle, \mathcal{S}(\{(\mathsf{skip}, 1)\})) \curvearrowright_{p_1} \langle c_1, \mathbb{R}_1, \mathbf{V}_1\rangle \Big\} \\ \mathsf{step}^{\mathcal{P}}_{\ell}(c_0, \mathcal{S}, j+1) = \Big\{ \big(\langle c_2, \mathbb{R}_2, \mathbf{V}_2\rangle, p_0 \cdot p_1\big) \;\Big|\; \\ \quad \big(\langle c_1, \mathbb{R}_1, \mathbf{V}_1\rangle, p_0\big) \in \mathsf{step}^{\mathcal{P}}_{\ell}(c_0, \mathcal{S}, j); \\ \quad (\langle c_1, \mathbb{R}_1, \mathbf{V}_1\rangle, \mathcal{S}(\mathbb{R}_1)) \curvearrowright_{p_1} \langle c_2, \mathbb{R}_2, \mathbf{V}_2\rangle \Big\} \end{array}$$

Figure 7: Real Clio Low Step Semantics

one. Rule Fetch-Exists is used when the sequence of interactions drawn produces a store that has a serialized labeled value indexed by $\underline{v}_k$ 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 $\underline{v}_k$, 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 has a labeled value that can be deserialized correctly, but whose recorded index is not the same as the index requested by the Clio computation or whose version is less than the version last seen.

Similar to the ideal store semantics, we use a low step relation $\curvearrowright_p$ to model adversary interactions, shown in Figure 7. The low step relation is also probabilistic as it is based on the probabilistic single step relation $\rightsquigarrow_p$. Additionally, we use a distribution of sequences of adversarial interactions $\mathbb{R}_A$ to model an adversary that behaves probabilistically. In rules Low-Step and Low-To-High-To-Low-Step a new distribution of interactions, $\mathbb{R}'$ is created by concatenating interaction sequences drawn from the existing distribution of interactions $\mathbb{R}$ and the adversary distribution $\mathbb{R}_A$. This is analogous to the application of adversary interactions to the current store in the ideal semantics. The rest of the definitions of the rules follow the same pattern as the ideal Clio low step store semantics.

With the low step relation, we use metafunction step to describe the distributions of real Clio configurations resulting from taking j low steps from configuration c_0, formally defined in Figure 7. The step function is parameterized by the keystore $\mathcal{P}$ and store level ℓ. To provide a source of adversary interactions while running the program, the step function also takes as input a *strategy* $\mathcal{S}$ which

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 $\mathcal{S}$ expresses the ability of the attacker to modify the store. The attacker chooses $\mathcal{S}$ (and t, v_0, and v_1), and $\mathcal{S}$ interacts with the store during execution. $\mathcal{S}$ 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 $\mathcal{S}$ to fetch values to determine the next modification to the store since $\mathcal{S}$ 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 $\overline{R_b}$, 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 $\mathcal{A}$,

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

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 $\mathrm{IND}_b(\mathcal{A}, n)$ denote the output of the experiment, where $\mathcal{A}$ is non-uniform ppt, $n \in \mathbb{N}$, $b \in \{0, 1\}$:

$$\begin{aligned}\mathrm{IND}_b(\mathcal{A}, n) = {} & (pk, sk) \leftarrow \mathit{Gen}(1^n);\\ & m_0, m_1, \mathcal{A}_2 \leftarrow \mathcal{A}(pk) \textit{ s.t. } |m_0| = |m_1|;\\ & c \leftarrow \mathit{Enc}(pk, m_b);\\ & \text{Output } \mathcal{A}_2(c)\end{aligned}$$

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

$$\left\{ \mathrm{IND}_0(\mathcal{A}, n) \right\}_n \approx \left\{ \mathrm{IND}_1(\mathcal{A}, n) \right\}_n$$

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 $\mathrm{IND}_b(\mathcal{P}, \mathcal{A}, \tilde{p}, j, n)$ denote the output of the following experiment, where $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Sign}, \mathsf{Verify})$, $\mathcal{A}$ is non-uniform ppt, $n \in \mathbb{N}$, $b \in \{0, 1\}$:

$$\begin{aligned}&\mathrm{IND}_b(\mathcal{P}_0, \mathcal{A}, \tilde{p}, j, n) =\\ &\quad \mathcal{P}' \leftarrow \mathsf{Gen}(\tilde{p}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\\ &\quad t, v_0, v_1, \mathcal{S}, \mathcal{A}_2 \leftarrow \mathcal{A}(\mathrm{pub}(\mathcal{P})) \text{ such that } v_0 =^C_\ell v_1\\ &\qquad\qquad \text{and } \vdash t : \mathsf{Labeled}\ \tau \to \mathsf{LIO}\ \tau'\\ &\qquad\qquad \text{and } \vdash v_0 : \mathsf{Labeled}\ \tau\\ &\qquad\qquad \text{and } \vdash v_1 : \mathsf{Labeled}\ \tau\\ &\qquad\qquad \text{and } \ell = \mathsf{authorityOf}(\mathcal{P}_0);\\ &\quad \langle c, \mathbb{R}_b, \mathbf{V}' \rangle \leftarrow \mathsf{step}^{\mathcal{P}}_\ell(\langle \mathsf{Start}(\mathcal{P}), \mathsf{Clr}(\mathcal{P}) \mid (t\ v_b) \rangle, \mathcal{S}, j);\\ &\quad \overline{R_b} \leftarrow \mathbb{R}_b;\ \text{Output } \mathcal{A}_2(\overline{R_b})\end{aligned}$$

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

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

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 ($\mathcal{P}_0$), the set of principals that Clio is protecting ($\tilde{p}$), and the number of low steps the program takes (j). Cryptosystem Π is used implicitly in the CTA game to generate keys, encrypt, decrypt, sign and verify[9].

[9] More formally, IND_b, $\mathcal{A}$, and the semantics are also parameterized on Π, and the uses of Gen, Enc, Dec, Sign, Verify should be explicitly taken from the tuple Π though we elide their explicit usage in our notation for clarity.

In this game setup, $\mathsf{Gen}(\tilde{p}, 1^n)$ generates a new keystore $\mathcal{P}'$ containing private keys for each of the principals in $\tilde{p}$, using the underlying cryptosystem's Gen function for each keypair. Then, the adversary receives all public keys of the keystore $\mathrm{pub}(\mathcal{P})$ and returns three well-typed Clio terms: a function t, and two program inputs to the function v_0 and v_1 that must be confidentiality-only low equivalent $=_\ell^C$ (i.e., they may differ only on secret values)[10]. It also returns a strategy $\mathcal{S}$ 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* Π *if CPA Secure, then Clio using* Π *is CTA Secure.*

We prove this theorem in part by induction over the low step relation $\curvearrowright_p$, 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 $\asymp$ on families of distributions of sequences of interactions preserves computational indistinguishability. That is, if $\mathbb{R}_1 \asymp \mathbb{R}_2$ and Π is CPA secure, then $\mathbb{R}_1 \approx \mathbb{R}_2$. Second, we show that as two low equivalent configurations step using the low step relation $\curvearrowright_p$, low equivalence is preserved and the interactions they produce satisfy the relation $\asymp$. 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 Clio store semantics, i.e., they merely describe the form of arbitrary interactions that may or may not have come from 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 Clio store semantics to break the circularity.

We progressively define the relation $\asymp$ 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 $l_1, ..., l_k$, such that $\mathbb{C}(l_i) \sqsubseteq^C \mathbb{C}(\mathrm{authorityOf}(\mathcal{P}))$, and for all $m^1_{\{1,2\}} ... m^n_{\{1,2\}}$ and all principals $\tilde{p}$, if $|m_1^i| = |m_2^i|$ for all $1 \le i \le k$ and Π is CPA Secure, then

$$\left\{ \begin{array}{l} \mathsf{put}\ \langle b_1^1 : l^1 \rangle\ \mathsf{at}\ \underline{v}^1 \cdot\ \ldots\ \cdot\ \mathsf{put}\ \langle b_1^k : l^k \rangle\ \mathsf{at}\ \underline{v}^k \ \Big|\ \\ \quad \mathcal{P} \leftarrow \mathsf{Gen}(1^n); \\ \quad (pk^i, sk^i) \in rng(\mathcal{P}); \\ \quad b_1^i \leftarrow \mathsf{Enc}(pk^i, m_1^i);\ 1 \le i \le k \end{array} \right\}_n$$

$$\asymp$$

$$\left\{ \begin{array}{l} \mathsf{put}\ \langle b_2^1 : l^1 \rangle\ \mathsf{at}\ \underline{v}^1 \cdot\ \ldots\ \cdot\ \mathsf{put}\ \langle b_2^k : l^k \rangle\ \mathsf{at}\ \underline{v}^k \ \Big|\ \\ \quad \mathcal{P} \leftarrow \mathsf{Gen}(1^n); \\ \quad (pk^i, sk^i) \in rng(\mathcal{P}); \\ \quad b_2^i \leftarrow \mathsf{Enc}(pk^i, m_2^i);\ 1 \le i \le k \end{array} \right\}_n$$

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 definiton of $\asymp$, we expand it to also contain interactions from a strategy, forming the final relationship on interactions captured by the $\asymp$ 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 $\asymp$ 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 $t =_\ell^C t'$ and the distributions of sequences of interactions they produce $\mathbb{R}$ and $\mathbb{R}'$ are in the relation $\asymp$.

We first show that if $c_0 \xrightarrow{\alpha} c_0'$ and $c_1 \xrightarrow{\alpha} c_1'$ and $c_0 =_\ell^C c_1$ then $c_0' =_\ell^C c_1'$. This proof takes advantage of the low equivalence preservation proofs for 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

$$(\langle c_0, \mathbb{R}_0, \mathbf{V}_0 \rangle, \mathbb{R}) \curvearrowright_p \langle c_0', \mathbb{R}_0', \mathbf{V}_0' \rangle$$

and

$$(\langle c_1, \mathbb{R}_1, \mathbf{V}_1 \rangle, \mathbb{R}) \curvearrowright_p \langle c_1', \mathbb{R}_1', \mathbf{V}_1' \rangle$$

and

$$(c_0 =_\ell^C c_1) \wedge (\mathbf{V}_0 = \mathbf{V}_1) \wedge (\mathbb{R}_0 \asymp \mathbb{R}_1)$$

[10] Complete definition of low equivalence is in the technical report [59].

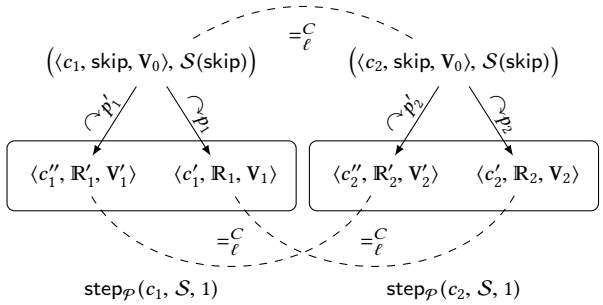

Figure 8: Low equivalence is preserved in $\text{step}_{\mathcal{P}}$ for two low equivalent configurations c_1 and c_2 and a strategy $\mathcal{S}$.

then,

$$(c'_0 =^C_\ell c'_1) \wedge (\mathbf{V}'_0 = \mathbf{V}'_1) \wedge (\mathbb{R}'_0 \asymp \mathbb{R}'_1).$$

The proof on $\curvearrowright_p$ relies on the previous preservation proof on $\xrightarrow{\alpha}$ and the indistinguishability results on $\asymp$.

Step 3: Indistinguishability of the step metafunction. We show that the step metafunction preserves low equivalence. More formally, we show that if $c_0 =^C_\ell c_1$ and $\mathbf{V}_0 = \mathbf{V}_1$ and $\mathbb{R}_0 \asymp \mathbb{R}_1$ then

$$\{(\mathbb{R}'_0, p_0 \cdot \ldots \cdot p) \mid \langle c_0, \mathbb{R}_0, \mathbf{V}_0\rangle \curvearrowright_{p_0} \ldots \curvearrowright_p \langle c'_0, \mathbb{R}'_0, \mathbf{V}'_0\rangle\}_n$$
$$\approx$$
$$\{(\mathbb{R}'_1, p'_0 \cdot \ldots \cdot p') \mid \langle c_1, \mathbb{R}_1, \mathbf{V}_1\rangle \curvearrowright_{p'_0} \ldots \curvearrowright_{p'} \langle c'_1, \mathbb{R}'_1, \mathbf{V}'_1\rangle\}_n$$

We prove this 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 result of $\text{step}_{\mathcal{P}}(c_1, \mathcal{S}, 1)$ and $\text{step}_{\mathcal{P}}(c_2, \mathcal{S}, 1)$ where $c_1 =^C_\ell c_2$. (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 c_1 configuration. The right circle shows the resulting distribution over distributions of configurations after one step of the c_2 configuration. Due to the results from Step 2, we can reason that $c'_1 =^C_\ell c'_2$ and that $c''_1 =^C_\ell c''_2$. We can also conclude that $\mathbb{R}_1 \asymp \mathbb{R}_2$ and that $\mathbb{R}'_1 \asymp \mathbb{R}'_2$. 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 p_1 is equal to p_2 and also p'_1 is equal to p'_2 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 Clio computation to illegitimately produce a signed value.

A digital signature scheme is secure if it is difficult to forge signatures of messages. Clio 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 Clio, 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 $\mathcal{P}_0$, non-uniform ppts $\mathcal{A}$, and labels l_1, integers j, j', where $\ell = \text{authorityOf}(\mathcal{P}_0)$ and $\mathbb{I}(l_1) \sqsubseteq^I p$, if Π is secure against existential forgery under chosen-message attacks, then*

$$\mathbf{Pr}\Big[\ \langle b{:}l_1\rangle \in \text{Values}_{\mathcal{P}}(\overline{R'}) \text{ and } \langle b{:}l_1\rangle \notin \text{Values}_{\mathcal{P}}(\overline{R}) \ \Big|\ \mathcal{P}' \leftarrow \text{Gen}(\{p\}, 1^n);\ \mathcal{P} = \mathcal{P}_0 \uplus \mathcal{P}';\ t, \mathcal{S}, \mathcal{A}_2 \leftarrow \mathcal{A}(\text{pub}(\mathcal{P}));\ \langle c, \mathbb{R}, \mathbf{V}\rangle \leftarrow \text{step}^{\mathcal{P}}_\ell(\langle \text{Start}(\mathcal{P}), \text{Clr}(\mathcal{P}) \mid t\rangle, \mathcal{S}, j);\ \overline{R} \leftarrow \mathbb{R};\ t', \mathcal{S}' \leftarrow \mathcal{A}_2(\overline{R});\ \langle c', \mathbb{R}', \mathbf{V}'\rangle \leftarrow \text{step}^{\mathcal{P}}_\ell(\langle \text{Start}(\mathcal{P}_0), \text{Clr}(\mathcal{P}) \mid t'\rangle, \mathcal{S}', j);\ \overline{R'} \leftarrow \mathbb{R}' \Big]$$

Intuitively, the game is structured as follows. First, an adversary chooses a term t and strategy $\mathcal{S}$ that will be run with high integrity (i.e., $\text{Start}(\mathcal{P})$ where $\mathcal{P}$ has p's authority). The adversary sees the interactions $\overline{R}$ 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 $\mathcal{S}'$ that will be run with low integrity (i.e., $\text{Start}(\mathcal{P}_0)$). 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 $\langle b{:}l_1\rangle$ that did not exist in the first run. In the experiment, the $\text{Values}_{\mathcal{P}}$ metafunction extracts the set of valid labeled bitstrings (i.e., can be deserialized correctly) using the parameterized keystore $\mathcal{P}$ 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 $\langle \text{Start}(\mathcal{P}_0), \text{Clr}(\mathcal{P}) \mid t\rangle$) cannot produce a high integrity value (i.e., a labeled value $\langle b{:}l\rangle$ such that $\mathbb{I}(l) \sqsubseteq^I p$).

6 Clio in Practice

6.1 Implementation

We implemented a Clio prototype as a Haskell library, in the same style as LIO. Building on the LIO code base, the Clio library has an API for defining and running Clio 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** Clio 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** Clio 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** Clio 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 primitives, we leverage the third-party cryptonite [27] library.

Clio 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 Clio 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 Clio 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 Clio computations `customerCode`, `preparerCode` and `irsCode`, respectively. We assume that the store level ℓ restricts writes to the store in confidential contexts, i.e. $\ell = \langle \bot, \top, S \rangle$, 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, `tpi`. The first step is to label `tpi` with the label $\langle C \vee P \vee \text{IRS}, C, S \rangle$. 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 the 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 Clio 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 $C \vee P \vee \text{IRS}$, 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 $l_1 = \langle P \vee \text{IRS}, P \vee C, S \rangle$. The entry in the database is labeled differently with $l_2 = \langle C \vee P \vee \text{IRS}, C, S \rangle$, but the operation succeeds because $l_2 \sqsubseteq l_1$ and the availability in l_2 is S, 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 raising 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_1, 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 ℓ, 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

```
customerCode :: TaxpayerInfo → CLIO ()
customerCode  tpi = do
    info ← label ⟨C ∨ P ∨ IRS, C, S⟩ tpi
    store "taxpayer_info" info
    return ()
```

```
preparerCode :: CLIO ()
preparerCode = do
    default ← label ⟨P ∨ IRS, P ∨ C, S⟩ notFound
    info ← fetch τ "taxpayer_info" default
    r ← toLabeled ⟨P ∨ IRS, P ∨ C, S⟩ $ do
        i ← unlabel info
        return (prepareTaxes i)
    store "tax_return" r
```

```
irsCode :: CLIO Bool
irsCode = do
    let l = ⟨IRS, P ∨ C ∨ IRS, S⟩
    default ← label l emptyTR
    lv ← fetch τ "tax_return" default
    tr ← unlabel lv
    return (verifyReturn tr)
```

Figure 9: Customer code (left), Preparer code (middle), and IRS code (right)

result, namely $\langle \text{IRS}, P \vee C \vee \text{IRS}, S \rangle$, which in this case is once again more restrictive than what is stored in the database. Thereafter the labeled tax return gets unlabeled 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 cryptography 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.

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 though access patterns. We force communication with the store to occur in non-sensitive contests. 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, and 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

[1] Ross Anderson and Roger Needham. 1995. Robustness Principles for Public Key Protocols. In *Annual International Cryptology Conference*. 236–247.

[2] Owen Arden, Michael D. George, Jed Liu, K. Vikram, Aslan Askarov, and Andrew C. Myers. 2012. Sharing Mobile Code Securely with Information Flow Control. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. 191–205.

[3] Owen Arden, Jed Liu, and Andrew C. Myers. 2015. Flow-Limited Authorization. In *Proceedings of the IEEE 28th Computer Security Foundations Symposium*. 569–583.

[4] Aslan Askarov and Stephen Chong. 2012. Learning is Change in Knowledge: Knowledge-based Security for Dynamic Policies. In *Proceedings of the IEEE Computer Security Foundations Symposium*.

[5] Aslan Askarov, Daniel Hedin, and Andrei Sabelfeld. 2006. Cryptographically-Masked Flows. In *Proceedings of the 13th International Static Analysis Symposium*.

[6] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. 2008. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*.

[7] Aslan Askarov and Andrei Sabelfeld. 2007. Gradual Release: Unifying Declassification, Encryption and Key Release Policies. In *Proceedings of the IEEE Symposium on Security and Privacy*. 207–221.

[8] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. 2010. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*.

[9] D. Asonov. 2005. *Querying Databases Privately: A New Approach to Private Information Retrieval*. Springer.

[10] Michael Backes, Dennis Hofheinz, and Dominique Unruh. 2009. CoSP: A General Framework for Computational Soundness Proofs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. 66–78.

[11] Michael Backes, Aniket Kate, Matteo Maffei, and Kim Pecina. 2012. ObliviAd: Provably Secure and Practical Online Behavioral Advertising. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*.

[12] John Bethencourt, Amit Sahai, and Brent Waters. 2007. Ciphertext-policy attribute-based encryption. In *Proc. of the 2007 IEEE Symposium on Security and Privacy*. 321–334.

[13] Niklas Broberg and David Sands. 2010. Paralocks: Role-based Information Flow Control and Beyond. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

[14] Winnie Cheng, Dan R.K. Ports, David Schultz, Victoria Popic, Aaron Blankstein, James Cowling, Dorothy Curtis, Liuba Shrira, and Barbara Liskov. 2012. Abstractions for Usable Information Flow Control in Aeolus. In *Proceedings of the 2012 USENIX Annual Technical Conference*. 139–151.

[15] Stephen Chong and Andrew C. Myers. 2006. Decentralized Robustness. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations*. 242–256.

[16] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan. 1995. Private Information Retrieval. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*.

[17] Tom Chothia, Dominic Duggan, and Jan Vitek. 2003. Type-Based Distributed Access Control. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*. 170–186.

[18] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: a web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and communications security*.

[19] Tim Dierks. 2008. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246. (2008). https://rfc-editor.org/rfc/rfc5246.txt

[20] Xuhua Ding, Yanjiang Yang, Robert H. Deng, and Shuhong Wang. 2010. A new hardware-assisted PIR with O(n) shuffle cost. *International Journal of Information Security* 9, 4 (2010).

[21] Cédric Fournet and Tamara Rezk. 2008. Cryptographically Sound Implementations for Typed Information-flow Security. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 323–335.

[22] Ivan Gazeau, Tom Chothia, and Dominic Duggan. 2017. Types for Location and Data Security in Cloud Environments. In *Proceedings of the IEEE Computer Sec. Foundations Symposium*.

[23] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. 2012. Hails: Protecting Data Privacy in Untrusted Web Applications. In *Proceedings of the Symposium on Operating Systems Design and Implementation*.

[24] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *Proceedings of the IEEE Symposium on Security and Privacy*. 11–20.

[25] Shafi Goldwasser and Mihir Bellare. 2001. *Lecture Notes on Cryptography*. Chapter 10.

[26] Shafi Goldwasser and Silvio Micali. 1982. Probabilistic Encryption & How to Play Mental Poker Keeping Secret All Partial Information. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*. 365–377.

[27] Vincent Hanquez. 2017. The cryptonite library. http://hackage.haskell.org/package/cryptonite. (2017).

[28] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proceedings of the 29th ACM Symposium on Applied Computing*.

[29] Redis Labs. 2015. Redis. http://redis.io/. (2015).

[30] Peeter Laud. 2001. Semantics and Program Analysis of Computationally Secure Information Flow. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*.

[31] Peeter Laud. 2008. On the Computational Soundness of Cryptographically Masked Flows. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 337–348.

[32] Peng Li, Yun Mao, and Steve Zdancewic. 2003. Information integrity policies. In *Proceedings of the Workshop on Formal Aspects in Security and Trust*.

[33] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. 2009. Fabric: A Platform for Secure Distributed Computation and Storage. In

Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles.

[34] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye, and Andrew C. Myers. 2009. Fabric: A Platform for Secure Distributed Computation and Storage. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. 321–334.

[35] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. 2013. PHANTOM: Practical Oblivious Computation in a Secure Processor. In *Proceedings of the ACM SIGSAC Conference on Computer & Communications Security*.

[36] John McLean. 1990. Security Models and Information Flow. In *Proceedings of the IEEE Symposium On Security And Privacy*. 180–187.

[37] Michael Mimoso. 2015. D-Link Accidentally Leaks Private Code-Signing Keys. https://threatpost.com/d-link-accidentally-leaks-private-code-signing-keys/114727/. (Sept. 2015).

[38] Pedro Moreno-Sanchez, Aniket Kate, Matteo Maffei, and Kim Pecina. 2015. Privacy Preserving Payments in Credit Networks: Enabling trust with privacy in online marketplaces. In *Proceedings of the Network and Distributed System Security Symposium*.

[39] Andrew C. Myers and Barbara Liskov. 1998. Complete, Safe Information Flow with Decentralized Labels.. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[40] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. 2006. Enforcing Robust Declassification and Qualified Robustness. *Journal of Computer Security* 14, 2 (April 2006), 157–196.

[41] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. 2001–. Jif: Java Information Flow. (2001–). Software release. http://www.cs.cornell.edu/jif.

[42] Femi G. Olumofin and Ian Goldberg. 2011. Revisiting the Computational Practicality of Private Information Retrieval. In *Proceedings of the 15th International Conference on Financial Cryptography and Data Security*.

[43] Rafael Pass and Abhi Shelat. 2010. *A Course in Cryptography* (3rd ed.). Chapter 7.

[44] Falko Peters. 2017. The hedis library. http://hackage.haskell.org/package/hedis. (2017).

[45] Ronald L Rivest, Adi Shamir, and Yael Tauman. 2006. How to leak a secret: Theory and applications of ring signatures. In *Theoretical Computer Science*. 164–186.

[46] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. 2009. Laminar: Practical Fine-grained Decentralized Information Flow Control. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[47] Alejandro Russo. 2015. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. 280–288.

[48] Rafia Shaikh. 2015. Microsoft Accidentally Leaks Xbox Live Keys, User Data at Risk of Man-in-the-Middle Attacks. http://wccftech.com/microsoft-accidentally-leaks-xbox-live-keys/. (Dec. 2015).

[49] V. Simonet. 2003. The Flow Caml System. (July 2003). Software release at http://cristal.inria.fr/~simonet/soft/flowcaml/.

[50] Radu Sion and Bogdan Carbunar. 2007. On the Computational Practicality of Private Information Retrieval. In *Proceedings of the Network and Distributed Systems Security Symposium. Stony Brook Network Security and Applied Cryptography Lab Tech Report*.

[51] Geoffrey Smith and Rafael Alpízar. 2006. Secure Information Flow with Random Assignment and Encryption. In *Proceedings of the 4th ACM Workshop on Formal Methods in Security*. 33–44.

[52] S. W. Smith and D. Safford. 2001. Practical Server Privacy with Secure Coprocessors. *IBM Systems Journal* 40, 3 (March 2001).

[53] Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. 2011. Disjunction Category Labels. In *Proceedings of the 16th Nordic Conference on Security IT Systems*. 223–239.

[54] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. 2011. Flexible Dynamic Information Flow Control in Haskell. In *Proceedings of the 4th ACM symposium on Haskell*. 95–106.

[55] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. 2014. Protecting Users by Confining JavaScript with COWL. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation*.

[56] Jeffrey A. Vaughan and Steve Zdancewic. 2007. A Cryptographic Decentralized Label Model. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*. 192–206.

[57] Veracode. 2015. *State of Software Security*. Vol. 6.

[58] Shuhong Wang, Xuhua Ding, Robert H. Deng, and Feng Bao. 2006. Private Information Retrieval Using Trusted Hardware. In *Proceedings of the 11th European Symposium on Research in Computer Security*. 49–64.

[59] Lucas Waye, Pablo Buiras, Owen Arden, Alejandro Russo, and Stephen Chong. 2017. Cryptographically Secure Information Flow Control on Key-Value Stores. *ArXiv e-prints* (Aug. 2017). arXiv:1708.08895

[60] Lucas Waye, Pablo Buiras, Dan King, Stephen Chong, and Alejandro Russo. 2015. It's My Privilege: Controlling Downgrading in DC-Labels. In *Proceedings of the 11th International Workshop on Security and Trust Management*.

[61] Alma Whitten and J. D. Tygar. 1999. Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. In *Proceedings of the 8th Conference on USENIX Security Symposium*.

[62] Peter Williams and Radu Sion. 2008. Usable PIR. In *Proceedings of the Network and Distributed System Security Symposium*.

[63] Alexander Yip, Neha Narula, Maxwell Krohn, and Robert Morris. 2009. Privacy-preserving browser-side scripting with BFlow. In *EuroSys*.

[64] Steve Zdancewic and Andrew C. Myers. 2001. Robust Declassification. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*. 15–23.

[65] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. 2008. Securing Distributed Systems with Information Flow Control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. 293–308.

[66] Lantian Zheng and Andrew C. Myers. 2005. End-to-End Availability Policies and Noninterference. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop*. 272–286.