2013 IEEE 26th Computer Security Foundations Symposium

Oblivious program execution and path-sensitive non-interference

Jérémy Planul
Stanford University

John C Mitchell
Stanford University

***Abstract*—Various cryptographic constructions allow an untrusted cloud server to compute over encrypted data, without decrypting the data. However, this prevents the cloud server from branching according to encrypted values. We study the constraints imposed by this important scenario by formulating and solving an equivalent information-flow problem, based on assuming an adversary could observe the control path. We develop a type system that prevents control-path information leaks, prove soundness, and compare with traditional implicit information-flow. Because simply preventing programs that leak information severely restricts the language, we define alternate (and easily implemented) semantics that execute multiple paths and combine the results using data operations. This produces a termination problem which we address with a more refined type system that characterizes a useful class of *obliviously executable* programs. We prove fundamental results about this language, semantics, and type system and conclude by comparing with traditional timing-based information-flow.**

I. Introduction

Cloud computing poses new security challenges [IDC, Chow et al., 2009] because programs may manipulate sensitive data on an untrusted or partially trusted platform. Modern cryptography such as homomorphic encryption [Gentry, 2009, Gentry, 2010, van Dijk et al., 2010] and secure multiparty computation [Ben-Or et al., 1988, Gennaro et al., 1998, Cramer et al., 2000, Naor and Nissim, 2001] may be used to compute on encrypted data without access to the decryption key (or associated process) that protects the data. However, if the program branches on an encrypted value, the cloud platform cannot determine which branch to execute because that depends on a hidden plaintext value. This language restriction can be formulated as an information-flow requirement: control-flow is public information and therefore cannot depend on private data.

In this paper, we formulate and study the information-flow problem associated with public branching on private data, which is distinct from traditional explicit and implicit information flow [Sabelfeld and Myers, 2003]. We develop a type system that prevents control-path information leaks, prove soundness, and compare with traditional implicit information-flow. Because simply preventing programs that leak information severely restricts the language, we define alternate (and appropriately implemented) semantics that execute multiple paths and combine the results using data operations. This produces a termination problem which we address with a more refined type system that characterizes a useful class of *obliviously executable* programs. We prove fundamental results about this language, semantics, and type system and conclude by comparing with traditional timing-based information-flow. This study produces an implemented language for computing over encrypted data that improves on previous solutions (*cf.* [Bain et al., 2011, Mitchell et al., 2012]).

As an illustrative setting providing both functional and imperative constructs, we work with the standard semantics of lambda calculus with references, augmented with path annotations representing the information available to an adversary that can tell which execution path is taken. Note that this problem of control-path information leaks has been studied before, with similar ideas, but with an imperative language [Molnar et al., 2006, Agat, 2000]. We consider standard information-flow typing concepts [Sabelfeld and Myers, 2003, Pottier and Simonet, 2003, Volpano et al., 1996] with security labels indicating high or low confidentiality. In a secure program, high-confidentiality values must not influence the control path or any observable result. This condition is formalized as a form of non-interference [Goguen and Meseguer, 1982].

Our initial candidate solution begins with a conventional information-flow type system (similar to the one of flow-caml [Pottier and Simonet, 2003]), for which we prove soundness for IO non-interference. We then extend the type system to prevent path information-flow leaks and prove soundness for path non-interference. Just as implicit information-flow can be reduced to explicit flow by a program transformation Sabelfeld and Myers [2003], we also define a program transformation that makes path leakage explicit, reducing path information-flow to IO information-flow.

Since simply preventing path information leakage reduces programming language expressiveness severely, we present an alternate semantics that effectively executes multiple paths and combines the results using data operations. This semantics is implementable with the correct information-flow properties (see, e.g., [Mitchell et al., 2012]) using homomorphic encryption or secure multiparty computation because such data operations hide the results of each path from the execution platform. We prove that this semantics has a weak confluence property we call *convergence,* implying that the semantics of a program is well-defined and equivalent in output to the standard semantics (but with

DOI 10.1109/CSF.2013.12

fewer path-flow leaks).

While the alternate semantics reduces path information leaks, it introduces another problem, namely termination of loops. If loop termination depends on a high confidentiality secret, then the loop *necessarily* fails to terminate because the semantics requires infinitely many paths to be executed. We solve this problem with a type system that allows arbitrary dependence on low confidentiality values, but only primitive recursion on high confidentiality values. This extends and improves on Mitchell et al. [2012], which proposes a similar but more restrictive language, explores its implementation with secure multiparty computation or homomorphic encryptions, but does not consider the more general language properties and analysis presented here. We prove that well-typed programs under this discipline terminate in the new semantics whenever they would terminate under the standard semantics and prove path-flow non-interference.

As a final contribution, we investigate the connection between path-flow information leakage and timing channels. We formalize timing leakage for our language and prove that our system preventing path-flow leakage also prevents timing leakage.

In Section II, we define our syntax and semantics. In Section III, we define our information-flow settings, and give some weak information-flow results (this section is standard and can be skipped by informed readers). In Section IV, we examine path information-flow problem, and propose our two solutions. In Section V, we acknowledge and solve our termination problem. In Section VI, we remark that we incidentally gave a solution for other implicit flows problems (e.g. timing channels). In Section VII, we show how our solution can be used for secure computation.

II. Programs and their semantics

In this section, we define our language and its semantics. The semantics has annotations that reveal the control-flow of the program (i.e., which path of the computation has been taken). We use those annotations in the sections below to formalize the leak of information.

A λ-calculus with references: The language is a typed call-by-value λ-calculus with booleans, an error value, conditionals, and references. We use the conditional as a way to express observable control-flow branching in a program, and we use the error value as a default value for uninitialized memory references. For general background on the λ-calculus, see Mitchell [1996], Pierce [2002].

$$\begin{array}{lcl} v & ::= & x \mid a \mid k \mid \bot \mid \lambda x.e \mid \mathsf{fix}\ f.\lambda x.e \\ e & ::= & v \mid v \wedge v \mid v \vee v \mid \neg v \mid e\ e \mid \mathsf{If}\ v\ \mathsf{then}\ e\ \mathsf{else}\ e \\ & & \mathsf{ref}\ e \mid !e \mid e := e \end{array}$$

where k range over 0, 1, and a range over a set Ref of references values. Values consist of variables, references, booleans, an error value, or lambda abstractions.

$$\begin{array}{rcll} \langle k \wedge k', \mu\rangle & \to & \langle k'', \mu\rangle & \text{where } k'' = k \wedge k' \\ \langle k \vee k', \mu\rangle & \to & \langle k'', \mu\rangle & \text{where } k'' = k \vee k' \\ \langle \neg k, \mu\rangle & \to & \langle k', \mu\rangle & \text{where } k' = \neg k \\ \langle (\lambda x.e)v, \mu\rangle & \to & \langle e[v/x], \mu\rangle & \\ \langle (\mathsf{fix}\ f.\lambda x.e)v, \mu\rangle & \to & \langle e[v/x][(\mathsf{fix}\ f.\lambda x.e)/f], \mu\rangle & \\ \langle \mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0, \mu\rangle & \xrightarrow{1} & \langle e_1, \mu\rangle & \text{when } v = 1 \\ \langle \mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0, \mu\rangle & \xrightarrow{0} & \langle e_0, \mu\rangle & \text{when } v = 0 \\ \langle \mathsf{ref}\ v, \mu\rangle & \to & \langle a, \mu\{a \mapsto v\}\rangle & \text{for } a \notin \mathsf{dom}(\mu) \\ \langle !v, \mu\rangle & \to & \langle \mu(v), \mu\rangle & \text{when } v \in \mathsf{dom}(\mu) \\ \langle !v, \mu\rangle & \to & \langle \bot, \mu\rangle & \text{when } v \notin \mathsf{dom}(\mu) \\ \langle v := v', \mu\rangle & \to & \langle 0, \mu\{v \mapsto v'\}\rangle & \text{when } v \in \mathsf{dom}(\mu) \\ \langle v := v', \mu\rangle & \to & \langle \bot, \mu\rangle & \text{when } v \notin \mathsf{dom}(\mu) \end{array}$$

Figure 1. Operational semantics

We write $\mathbb{V}$ for the set of values. Expressions consist of values, conjunctions, disjunctions, negations, functions applications, conditionals, references allocations, dereferences, or assignments. We adopt standard notions of free and bound variables, of closed expressions, and of the capture-avoiding substitution $e[e'/x]$ of an expression e' for all free occurrences of a variable x in an expression e. We write $e_1; e_2$ as syntactic sugar for $(\lambda x.e_2)\ e_1$ where x is not free in e_2. We write $\mathsf{If}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0$ as syntactic sugar for $(\lambda x.\mathsf{If}\ x\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0)\ e$ where x is not free in e_0 and e_1. Intuitively, the error value $\bot$ can have any type, and direct computations with an error results in an error. However, only branches taken by the expression can result in errors. For instance $\mathsf{If}\ 1\ \mathsf{then}\ e\ \mathsf{else}\ \bot$ does not result in an error. The reduction of $\mathsf{If}\ v\ \mathsf{then}\ e\ \mathsf{else}\ e$ is annotated with the value v, as a visible path-flow. In the following, we only consider values and expressions that are simply typed (we omit the standard λ-calculus typing rules).

Semantic: A *memory* is a mapping $\mu : \mathrm{Ref} \to \mathbb{V}$ that sends references to values of our language. A *configuration* is a pair $\langle e, \mu\rangle$ with μ a memory and e an expression. The small step semantics is a labeled transition relation $\xrightarrow{\alpha}$ (where α is 0, 1, or nothing). Basic rules for the semantics are defined in Figure 1 (with $\mathsf{dom}(\mu) = \{a | \mu(a) \neq \bot\}$) and context rules are defined in Figure 2. (We omit the rules of computation with $\bot$.) Most of these rules are standard except the rules for the conditional, which annotate the transition with a value corresponding to the path taken. Hence, the control-flow of the program execution is visible on the trace. Those trace annotations are a model for when the control-flow is visible by the host.

Example 1: Let a be a reference, we consider the expression $\mathsf{If}\ !a\ \mathsf{then}\ 0\ \mathsf{else}\ 0$, and we consider the two initial memories $\mu_0 = [a \mapsto 0]$ and $\mu_1 = [a \mapsto 1]$. We have

$$\langle \mathsf{If}\ !a\ \mathsf{then}\ 0\ \mathsf{else}\ 0, \mu_0\rangle \xrightarrow{0} \langle 0, \mu_0\rangle$$

$$\langle \mathsf{If}\ !a\ \mathsf{then}\ 0\ \mathsf{else}\ 0, \mu_1\rangle \xrightarrow{1} \langle 0, \mu_1\rangle$$

The two final results are equal, but the two traces are different.

$$\frac{\langle e_1,\mu\rangle \xrightarrow{\alpha} \langle e_1',\mu'\rangle}{\langle e_1 e_2,\mu\rangle \xrightarrow{\alpha} \langle e_1' e_2,\mu'\rangle} \qquad \frac{\langle e_2,\mu\rangle \xrightarrow{\alpha} \langle e_2',\mu'\rangle}{\langle v_1 e_2,\mu\rangle \xrightarrow{\alpha} \langle v_1 e_2',\mu'\rangle}$$

$$\frac{\langle e,\mu\rangle \xrightarrow{\alpha} \langle e',\mu'\rangle}{\langle \mathsf{ref}\ e,\mu\rangle \xrightarrow{\alpha} \langle \mathsf{ref}\ e',\mu'\rangle} \qquad \frac{\langle e,\mu\rangle \xrightarrow{\alpha} \langle e',\mu'\rangle}{\langle !e,\mu\rangle \xrightarrow{\alpha} \langle !e',\mu'\rangle}$$

$$\frac{\langle e_1,\mu\rangle \xrightarrow{\alpha} \langle e_1',\mu'\rangle}{\langle e_1 := e_2,\mu\rangle \xrightarrow{\alpha} \langle e_1' := e_2,\mu'\rangle}$$

$$\frac{\langle e_2,\mu\rangle \xrightarrow{\alpha} \langle e_2',\mu'\rangle}{\langle v_1 := e_2,\mu\rangle \xrightarrow{\alpha} \langle v_1 := e_2',\mu'\rangle}$$

Figure 2. Context rules

We write $\langle e,\mu\rangle \xrightarrow{\gamma}{}^{*} \langle e',\mu'\rangle$ with $\gamma = \alpha_0\alpha_1\ldots\alpha_n$ for the corresponding big-step semantics, i.e., when

$$\langle e,\mu\rangle \xrightarrow{\alpha_0} \langle e_1,\mu_1\rangle \xrightarrow{\alpha_1} \ldots \xrightarrow{\alpha_n} \langle e',\mu'\rangle$$

for some $\alpha_0,\alpha_1,\ldots,\alpha_n$ and $e,e_1,\ldots,e'$. When there exists no α, e', and μ' such that $\langle e,\mu\rangle \xrightarrow{\alpha} \langle e',\mu'\rangle$; we say that $\langle e,\mu\rangle$ is irreducible (in this semantics, this appends when e is a value).

III. IO Information-Flow

We now discuss our security policies. We annotate our types with security labels, and consider the security of our programs given an environment that specifies the types of the free references. This specifies the security intent; for instance, free references with a private type are supposed not to influence public references (or in the case of path non-interference, the control-flow). We then define the corresponding security statement in terms of different types of non-interference (IO and path non-interference) and see how this control-flow leak is detrimental to security. If we exclude the definition of path non-interference, this section is mostly standard.

Security types: We first discuss our security types:

$$\begin{array}{lcl} \ell & ::= & L \mid H \\ t & ::= & \mathit{bool} \mid \tau\ \mathsf{ref} \mid (\tau \xrightarrow{\ell} \tau) \\ \tau & ::= & t^{\ell} \end{array}$$

We consider public (L) and private (H) security labels, and a reflexive relation $\leq$ such that $L \leq H$ but $H \not\leq L$ ($\sqcup$ and $\sqcap$ are the associated least upper bound and greatest lower bound on security labels). *Types* (t) consist of booleans, references, and abstractions. References recursively contain the security type of the value they refer to. Abstractions specify the security type of the value they take as input, the security type of the resulting application, and an additional label to specify their pc, i.e. a lower bound on the label of the references they modify. For instance, a function of type $\tau \xrightarrow{H} \tau'$ is guaranteed to never write on public references. *Security types* (τ) are labeled types, for instance a value of type $(\mathit{bool}^H\ \mathsf{ref})^L$ is public reference to a confidential boolean. We extend $\leq$ on security types by saying that $\ell \leq t^{\ell'}$ when $\ell \leq \ell'$. We let $t^{\ell} \sqcup \ell'$ stands for $t^{\ell \sqcup \ell'}$.

Environment: We now define security policies, which give security types to the free references of the expression. More generally, we define *environments* Γ as functions from variables and references to security types (τ). These environments serve for initial security policy and are also used in our type system where they specify types for bound variables and references.

Memory typing: We want to consider expressions containing free reference names and running on initialized memories, hence we need a way to specify when a particular memory corresponds to the environment. When the environment states that a memory reference contains a boolean, it suffices to check that it is indeed the case. However, the environment may also state functional information-flow types or reference types. When this is the case, we check it with an information-flow typing judgment $\Gamma \vdash v : \tau$. Our judgment for memory typing is parametric in $\vdash$ (since typing judgments have not yet been defined). It is of the form $\Gamma \models \mu$, and is defined inductively as follows:

$$\frac{\Gamma' \models \mu \qquad \Gamma \vdash v : \tau \qquad v \neq \bot}{\Gamma'; a : \tau \models \mu[a \mapsto v]} \qquad \emptyset \models \forall a. a \mapsto \bot$$

For a given Γ, the first statement says that, if a memory μ is correctly typed for some partial environment Γ' (typically, Γ' is a subset of Γ), and an additional value v types (with judgment $\vdash$) in the initial environment Γ at level τ, then the memory enriched with this value at a reference a types in the environment Γ' enriched with reference a at type τ. The second statement ensures that memory references that don't appear in the environment contain the default error value. When $\Gamma \models \mu$ for Γ and $\vdash$, we say that μ is a typed memory for $\vdash$ and Γ. When it is clear from the context, we may omit $\vdash$ or Γ.

Low-equalities: Once a security policy is defined, it is possible to partially compare two memories relatively to this policy. For example, if we determine that two memory are equal for public references, it is not possible distinguish them by accessing only the public references. Formally, let Γ be an environment; let μ_0 and μ_1 be two memories. We say that $\mu_0 =_L \mu_1$ when, for every reference a such that $\Gamma(a) = t^L$, we have $\mu_0(a) = \mu_1(a)$. We also define a weaker equality that only examines boolean constants. We say that $\mu_0 =_{\mathit{bool}-L} \mu_1$ when, for every reference a such that $\Gamma(a) = \mathit{bool}^L$, we have $\mu_0(a) = \mu_1(a)$. This is useful to avoid some complications of non-interference in a functional setting (for instance when two functions are semantically equivalent, but syntactically not equal), which are not the focus of this article.

Non-interference: For IO non-interference (standard non-interference), an expression is considered secure when the initial values of the privates references do not influence the final values of the public references. In other words, an expression is secure when its execution on two initial memories with the sames values in public references leads to two final memories with the same values in public references. Additionally, we define path non-interference, in which the control-flow channel is considered public. In this case, an expression is considered secure when the initial values in privates references do not influence the final values in public references or the semantics path annotations.

Definition 1: Let Γ be an environment, ℓ a label, and e an expression. We say that e is IO non-interferent for $\vdash$ when, for every typed memories μ_0 and μ_1 such that $\mu_0 =_L \mu_1$, and for every expressions e_0 and e_1, memories μ_0' and μ_1', and traces γ_0 and γ_1 such that

$$\langle e, \mu_0 \rangle \xrightarrow{\gamma_0}{}^{*} \langle e_0, \mu_0' \rangle$$

$$\langle e, \mu_1 \rangle \xrightarrow{\gamma_1}{}^{*} \langle e_1, \mu_1' \rangle$$

and both $\langle e_0, \mu_0' \rangle$ and $\langle e_1, \mu_1' \rangle$ are irreducible, we have $\mu_0' =_{bool-L} \mu_1'$. If, additionally, $\gamma_0 = \gamma_1$, we say that e is *path* non-interferent.

We may omit $\vdash$ when it is obvious from the context.

Type system for non-interference: As a starting point, we equip our language with a type system that enforces IO non-interference. Typing judgments for values are of the form $\Gamma \vdash v : \tau$. Typing judgments for expressions are of the form $\ell, \Gamma \vdash e : \tau$. The typing rules for commands appear in Figure 3. This type system is similar to those typically used for non-interference [see e.g. Sabelfeld and Myers, 2003].

The judgment $\ell, \Gamma \vdash e : \tau$ guarantees that e is of type τ, that it only writes references with a level higher than ℓ (sometimes called its 'program counter' level), and that it is IO non-interferent. In the application rule, we match the input type of the abstraction with the type of the input, we also need to make sure that the level of the abstraction flows to the the pc of its arrow type. In the conditional rule, we ensure that the guard is not more confidential than the pc of the branches. In the assignment rule, we ensure that the types matches, and that the assigned reference is taken account of in the pc of the expression.

The judgment $\Gamma \vdash v : \tau$ guarantees that v is of type τ. Boolean constants can be seen as of type $bool^L$. Reference names and variables get their types from the environment. For abstractions, if we can type the abstracted expression in an environment with the abstraction variable at some type, we get an arrow type linking both types; the arrow is annotated with the pc used to type the expression, thus specifying a lower bound on the level of variables accessed when applying the abstraction.

Relations between types

$$\frac{\ell \leq \ell' \qquad t \leq t'}{t^{\ell} \leq t'^{\ell'}} \qquad \frac{\tau_0' \leq \tau_0 \qquad \tau_1 \leq \tau_1' \qquad \ell' \leq \ell}{\tau_0 \xrightarrow{\ell} \tau_1 \leq \tau_0' \xrightarrow{\ell'} \tau_1'}$$

Type judgments for values

$$\Gamma \vdash k : bool^L \qquad \Gamma \vdash \bot : \tau \qquad \frac{\Gamma \vdash v : \tau \qquad \tau \leq \tau'}{\Gamma \vdash v : \tau'}$$

$$\frac{\Gamma(a) = \tau}{\Gamma \vdash a : \tau\ \mathsf{ref}^L} \qquad \Gamma \vdash x : \Gamma(x) \qquad \frac{\ell, \Gamma; x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : (\tau \xrightarrow{\ell} \tau')^L}$$

$$\frac{\ell, \Gamma; x : \tau; f : (\tau \xrightarrow{\ell} \tau')^{\ell'} \vdash e : \tau'}{\Gamma \vdash \mathsf{fix}\ f.\lambda x.e : (\tau \xrightarrow{\ell} \tau')^{\ell'}}$$

Type judgments for expressions

$$\frac{\ell, \Gamma \vdash e : \tau \qquad \tau \leq \tau' \qquad \ell' \leq \ell}{\ell', \Gamma \vdash e : \tau'} \qquad \frac{\Gamma \vdash v : \tau}{H, \Gamma \vdash v : \tau}$$

$$\frac{\Gamma \vdash v : bool^{\ell} \qquad \Gamma \vdash v' : bool^{\ell}}{H, \Gamma \vdash v \wedge v' : bool^{\ell}}$$

$$\frac{\Gamma \vdash v : bool^{\ell} \qquad \Gamma \vdash v' : bool^{\ell}}{H, \Gamma \vdash v \vee v' : bool^{\ell}} \qquad \frac{\Gamma \vdash v : bool^{\ell}}{H, \Gamma \vdash \neg v : bool^{\ell}}$$

$$\frac{\ell, \Gamma \vdash e : (\tau' \xrightarrow{\ell'} \tau)^{\ell''} \qquad \ell, \Gamma \vdash e' : \tau' \qquad \ell'' \leq \ell'}{\ell \sqcap \ell', \Gamma \vdash e\ e' : \tau \sqcup \ell''}$$

$$\frac{\Gamma \vdash v : \mathsf{bool}^{\ell} \qquad \ell, \Gamma \vdash e_1 : \tau \qquad \ell, \Gamma \vdash e_0 : \tau}{\ell, \Gamma \vdash \mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0 : \tau \sqcup \ell}$$

$$\frac{\ell, \Gamma \vdash e : t^{\ell}}{\ell, \Gamma \vdash \mathsf{ref}\ e : t^{\ell}\ \mathsf{ref}^L} \qquad \frac{\ell, \Gamma \vdash e : t^{\ell'}\ \mathsf{ref}^{\ell''}}{\ell, \Gamma \vdash !e : t^{\ell' \sqcup \ell''}}$$

$$\frac{\ell, \Gamma \vdash e : t^{\ell'}\ \mathsf{ref}^{\ell'} \qquad \ell, \Gamma \vdash e' : t^{\ell'}}{\ell \sqcap \ell', \Gamma \vdash e := e' : bool^L}$$

Figure 3. Type system for IO non-interference

$$
\begin{array}{lcl}
[\![\mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0]\!]_b & = & \mathsf{If}\ v\ \mathsf{then}\ [\![e_1]\!]_b; b := 1 \\
 & & \quad\ \mathsf{else}\ [\![e_0]\!]_b; b := 0 \\
[\![e\ e']\!]_b & = & [\![e]\!]_b\ [\![e']\!]_b \\
[\![\lambda x.e]\!]_b & = & \lambda x.[\![e]\!]_b \\
[\![\mathsf{fix}\ f.\lambda x.e]\!]_b & = & \mathsf{fix}\ f.\lambda x.[\![e]\!]_b \\
[\![\mathsf{ref}\ v]\!]_b & = & \mathsf{ref}\ v \\
[\![!v]\!]_b & = & !v \\
[\![v := v']\!]_b & = & v := v' \\
[\![x]\!]_b & = & x
\end{array}
$$

Figure 4. Transformation for explicit leak

Theorem 1: Let Γ be an environment, let e be an expression, let τ be a security type, and ℓ a label such that $\ell, \Gamma \vdash e : \tau$. Then e is IO non-interferent.

IV. Path Information-Flow

The above type system does guarantee IO non-interference but not path non-interference. In the example below, we show a typed expression (hence IO non-interferent) that branches on a private reference, and thus is not path non-interferent.

Example 2: Let h be a reference. We consider the expression $\mathsf{If}\ !h\ \mathsf{then}\ 0\ \mathsf{else}\ 0$ and the environment $\Gamma = h : bool^H$. We have $H, \Gamma \vdash \mathsf{If}\ !h\ \mathsf{then}\ 0\ \mathsf{else}\ 0 : H$. However, if we consider the two low-equal memories $\mu_0 = [h \mapsto 0]$ and $\mu_1 = [h \mapsto 1]$, we have

$$\langle \mathsf{If}\ !h\ \mathsf{then}\ 0\ \mathsf{else}\ 0, \mu_0 \rangle \xrightarrow{0} \langle 0, \mu_0 \rangle$$

$$\langle \mathsf{If}\ !h\ \mathsf{then}\ 0\ \mathsf{else}\ 0, \mu_1 \rangle \xrightarrow{1} \langle 0, \mu_1 \rangle$$

The two final memories are low-equal, but the two traces are different; the expression is not path non-interferent.

Conditional leak the value of their guard in the trace. Hence, we can secure the type system by replacing the conditional rule with a rule that disallows private guards (as previously observed by Smith and Volpano [1998]):

$$\frac{\Gamma \vdash v : \mathsf{bool}^L \qquad L, \Gamma \vdash e_1 : \tau \qquad L, \Gamma \vdash e_0 : \tau}{L, \Gamma \vdash \mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0 : \tau}$$

The resulting type system guarantees path non-interference, but fewer expressions can be typed.

Theorem 2: Let Γ be an environment, let e be an expression, let τ be a security type, and ℓ a label such that $\ell, \Gamma \vdash e : \tau$ in the type system with the more restrictive conditional rule. Then e is path non-interferent.

Transformation: As an alternative, it is also possible to detect expressions that leak information through their control-flow by making the leak explicit. In Figure 4, we describe such a transformation, in which we add a new public reference that explicitly records the control-flow leak. In the transformed expression, the control-flow now affects our public reference; hence, if the transformed expression is non-interferent, then no private information influences the control-flow. Thus, the original expression must be path non-interferent. In the theorem below, we show that, if a transformed expression types in the standard information-flow type system (i.e., is IO non-interferent), then the original expression is path non-interferent.

Theorem 3: Let e be an expression, let b be a fresh reference name, let Γ be an environment such that $\Gamma(b) = bool^L$, let τ be a security type, and ℓ a label such that $\ell, \Gamma \vdash [\![e]\!]_b : \tau$ in the original type system. Then e is path non-interferent.

Still, this approach has the same effect of disallowing private guards, and accepts exactly the same reduced set of expressions than the type system with the restrictive conditional rule.

Bypassing control-flow leaking: Formally, it is sometimes possible to rewrite programs to avoid the leak itself. For instance, if the two branches of a conditional are boolean constants, it is possible to implement it using basic operations, thus denying the leak. For example, let v, k_1 and k_0 be booleans, $\mathsf{If}\ v\ \mathsf{then}\ k_1\ \mathsf{else}\ k_0$ gives the same result than $(v \wedge k_1) \vee (\neg v \wedge k_0)$ but avoids the control-flow leak. Instead of transforming the original expression, we can achieve a similar result in the semantics by adding a rule

$$\langle \mathsf{If}\ v\ \mathsf{then}\ k_1\ \mathsf{else}\ k_0, \mu \rangle \rightarrow (v \wedge k_1) \vee (\neg v \wedge k_0)$$

In practice, we can see the conditional (and its path leak) in our model as a way for the programmer to specify that requests on any of the two values is observable by the host (for instance, if they are stored on a remote machine). Hence, a request on one of the two values reveal the branch taken. In this case, the rewritten program requests both values, and thus avoid any path leak (similarly with the additional semantic rule). We can also draw parallels with homomorphic operations on encryptions. Homomorphic operations are computations done on encrypted plaintexts without reading the plaintexts themselves, and thus without revealing any information on those plaintexts. In such settings, it is not possible to do a generic conditional homomorphicaly since it would leak the control-flow, but it is possible to implement it on constants as above using basic operations.

This extended semantics works when the branches of a conditional are booleans, but not when the branches are expressions, references, or functions. In pure lambda calculus, the order of evaluation is irrelevant; with such a property, we could reduce the branches, and, in some cases, transform the conditional using the semantic rule above. However, this is not the case in lambda calculus with references since operations on references are only executed on the valid side of the conditional. Still, if we allow some expressions in memory (not only values), it is possible to have correct nested evaluations. For instance, when some reference a is updated on one side of a conditional, we replace the expression for a in the memory by a conditional expression on the updated value and the previous expression for a.

Even with nested evaluation, we still cannot safely (avoiding control-flow leaks) reduce conditionals of non-boolean types to values. However, we can adapt our semantics to reduce expressions containing such conditionals. For instance, if a dereference is applied on a conditional of references, we can safely distribute the dereference on the conditional. We can similarly safely distribute assignments and reference allocations. Similarly, when the branches values are lambda abstractions, we can safely factorize the abstraction over the conditional. Conditional trees of references are still irreducible (if we want to avoid control-flow leaks), hence, if we want to safely reduce all expressions that are not reference-typed to values, we need to allow the application of functions to those conditional trees of references (and not only values). Here is how we define those trees of references:

$$a \text{ is a tree of references (for all reference } a)$$

$$\frac{e_1, e_0 \text{ are trees of references}}{\mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0 \text{ is a tree of references}}$$

We now formally extend stores so that they can contain pure expressions, and not only values. We say that an expression e is *pure* when it contains no reference allocation, dereference, assignment, or application:

$$v \text{ is pure} \qquad \frac{e_1, e_0 \text{ are pure}}{\mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0 \text{ is pure}}$$

The main rule for store typing becomes

$$\frac{\Gamma' \models \mu \qquad \Gamma \vdash e : \tau \qquad e \text{ is pure}}{\Gamma; l : \tau \models \mu[l \mapsto e]}$$

Then, in Figure 5, we describe our extension to the operational semantics. In Rules (Cond-L) and (Cond-R), we describe how to safely reduce expressions inside a conditional. To this end, we check the result of one step of reduction in one of the expression, then we update the memory with conditionals referring the two possible states (depending on the conditional branch). Rule (Mem) is for reductions in memory. Rule (App) is for application on trees of references. Rule (Fact-λ) is for factoring lambda abstractions on conditionals. Rule (Fact-f) is for factoring fixpoints on conditionals. Rule (Distr-d) is for distributing dereferences on conditionals. Rule (Distr-al) is for distributing assignments on the right of conditionals. Rule (Distr-ar) is for distributing assignments on the left of conditionals. Rule (Distr-r) is for distributing references declaration on conditionals. Rule (No-flow) is for reducing conditionals in arithmetic operations.

Convergence: This new semantics is not deterministic anymore, but it is convergent on pure memories for expressions that reduce to booleans (expressions that reduce to functions might get different, although equivalents, results).

Theorem 4: Let μ be a pure memory. Let e be an expression. If $\langle e, \mu\rangle \xrightarrow{\gamma_0}{}^{*} \langle e_0, \mu_0\rangle$, then, μ_0 is pure. Also, if $\langle e, \mu\rangle \xrightarrow{\gamma_1}{}^{*} \langle e_1, \mu_1\rangle \xrightarrow{\gamma_1'}{}^{*} \langle k, \mu_1'\rangle$, then, there exists μ_0', and γ_0' such that $\langle e_0, \mu_0\rangle \xrightarrow{\gamma_0'}{}^{*} \langle k, \mu_0'\rangle$.

Our additional rules do not change the result of expressions, but they do induce different semantic annotations (they change the strategy of evaluation). In this example, we show how we can use those additional rules to avoid control-flow leaks.

Example 3: Let h be a reference name. We consider the following execution

$$\begin{aligned}
&\langle \mathsf{If}\ 0\ \mathsf{then}\ h := 1; 1\ \mathsf{else}\ h := 0; 0, \{h \mapsto 1\}\rangle \\
&\quad \to \langle \mathsf{If}\ 0\ \mathsf{then}\ 1\ \mathsf{else}\ h := 0; 0, \{h \mapsto \mathsf{If}\ 0\ \mathsf{then}\ 1\ \mathsf{else}\ 1\}\rangle \\
&\quad \to \langle \mathsf{If}\ 0\ \mathsf{then}\ 1\ \mathsf{else}\ 0, \\
&\qquad\qquad \{h \mapsto \mathsf{If}\ 0\ \mathsf{then}\ \mathsf{If}\ 0\ \mathsf{then}\ 1\ \mathsf{else}\ 1\ \mathsf{else}\ 0\}\rangle \\
&\quad \to^{*} \langle (0 \wedge 1) \vee (\neg 0 \wedge 0), \\
&\qquad\qquad \{h \mapsto (0 \wedge ((0 \wedge 1) \vee (\neg 0 \wedge 0))) \vee (\neg 0 \wedge 0)\}\rangle \\
&\quad \to^{*} \langle 0, \{h \mapsto 0\}\rangle
\end{aligned}$$

This execution is longer than an execution in the original semantics (since it executes both paths), but it gives the same result and has no annotations.

$$\begin{aligned}
&\langle \mathsf{If}\ 0\ \mathsf{then}\ h := 1; 1\ \mathsf{else}\ h := 0; 0, \{h \mapsto 1\}\rangle \\
&\qquad\qquad \xrightarrow{0} \quad \langle h := 0; 0, \{h \mapsto 1\}\rangle \\
&\qquad\qquad \to \quad \langle 0, \{h \mapsto 0\}\rangle
\end{aligned}$$

We avoid control-flow leaks of secret values when we use those extended rules on conditionals with a private guard, but we don't need to use them when the guard is public. In order to guide the usage of the extended rules (so that they are not used when the conditional is public), we add annotations on the conditionals. We consider the annotated conditionals If_L and If_H, named respectively H- and L-conditionals. From now on, we use exclusively the additional rules for H-conditionals, and exclusively the original rules for L-conditionals (except when specified otherwise). Note that this makes it so that some irreducible expressions (H-trees of references) are not values.

Theorem 5: Let μ be a pure memory. Let e be an expression. If $\langle e, \mu\rangle$ is irreducible, then e is a value, or e is a tree of references.

To satisfy standard notion of progress, we could consider H-trees of references as values.

V. Termination

We showed that our additional semantic rules are consistent; hence we can safely execute the program while avoiding the leaking conditional rule when the guard is private. However, since we now run additional branches, we have no guarantee that an expression that terminates with the initial rules still terminates with the extended rules, as shown in the example below.

Example 4: We consider the following expression

$$\mathsf{If}\ 0\ \mathsf{then}\ \Omega\ \mathsf{else}\ 1$$

$$\text{(Cond-L)}\ \frac{\langle e_1,\mu\rangle \xrightarrow{\alpha} \langle e_1',\mu'\rangle \qquad \mu''(a) = \mathsf{If}\ v\ \mathsf{then}\ \mu'(a)\ \mathsf{else}\ \mu(a)\ \text{for}\ a \in dom(\mu')}{\langle \mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0,\mu\rangle \xrightarrow{\alpha} \langle \mathsf{If}\ v\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_0,\mu''\rangle}$$

$$\text{(Cond-R)}\ \frac{\langle e_0,\mu\rangle \xrightarrow{\alpha} \langle e_0',\mu'\rangle \qquad \mu''(a) = \mathsf{If}\ v\ \mathsf{then}\ \mu(a)\ \mathsf{else}\ \mu'(a)\ \text{for}\ a \in dom(\mu')}{\langle \mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0,\mu\rangle \xrightarrow{\alpha} \langle \mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0',\mu''\rangle}$$

$$\text{(Mem)}\ \frac{\langle e,\mu\rangle \xrightarrow{\alpha} \langle e',\mu\rangle \qquad \mu(a) = e}{\langle e_0,\mu\rangle \xrightarrow{\alpha} \langle e_0,\mu[a \mapsto e']\rangle} \qquad\qquad \text{(App)}\ \frac{e'\ \text{is a tree of references}}{\lambda x.e\ e' \to e[e'/x]}$$

$$\begin{array}{lrcl}
\text{(Fact-}\lambda\text{)} & \langle \mathsf{If}\ v\ \mathsf{then}\ \lambda x.e_1\ \mathsf{else}\ \lambda x.e_0,\mu\rangle & \to & \langle \lambda x.\mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0,\mu\rangle \\
\text{(Fact-f)} & \langle \mathsf{If}\ v\ \mathsf{then}\ \mathsf{fix}\ f.\lambda x.e_1\ \mathsf{else}\ \mathsf{fix}\ f.\lambda x.e_0,\mu\rangle & \to & \langle \mathsf{fix}\ f.\lambda x.\mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0,\mu\rangle \\
\text{(Distr-d)} & \langle !\mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0,\mu\rangle & \to & \langle \mathsf{If}\ v\ \mathsf{then}\ !e_1\ \mathsf{else}\ !e_0,\mu\rangle \\
\text{(Distr-al)} & \langle (\mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0) := e_2,\mu\rangle & \to & \langle \mathsf{If}\ v\ \mathsf{then}\ e_1 := e_2\ \mathsf{else}\ e_0 := e_2,\mu\rangle \\
\text{(Distr-ar)} & \langle v := (\mathsf{If}\ v'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0),\mu\rangle & \to & \langle \mathsf{If}\ v'\ \mathsf{then}\ v := e_1\ \mathsf{else}\ v := e_0,\mu\rangle \\
\text{(Distr-r)} & \langle \mathsf{ref}\ (\mathsf{If}\ v'\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0),\mu\rangle & \to & \langle \mathsf{If}\ v'\ \mathsf{then}\ \mathsf{ref}\ e_1\ \mathsf{else}\ \mathsf{ref}\ e_0,\mu\rangle \\
\text{(No-flow)} & \langle \mathsf{If}\ k_0\ \mathsf{then}\ k_1\ \mathsf{else}\ k_2,\mu\rangle & \to & \langle (k_0 \wedge k_1) \vee (\neg k_0 \wedge k_2),\mu\rangle
\end{array}$$

Figure 5. Extension to the operational semantics

where Ω is a non-terminating expression. This expression terminates, but only with the original semantic rules.

This problem is inherent to the idea of not leaking control flow. Intuitively, if the secret values cannot influence which branches are executed, the execution must be at least as long as the longest possible execution; if the execution is not bounded by public values, the program cannot terminate. In order to avoid this problem, we require termination of the branches of H-conditionals, and, more generally, of all expressions typed with a secret pc. In typed lambda calculus with references, non-termination can appear with two mechanisms: fixpoints (which are easy to disallow) and an encoding of fixpoint using references, as in the following expression

$$a := (\lambda x.(!a\ 1)); !a\ 1$$

It is possible to disallow this kind of encoding by including the type of the references read in the type of functions. Then, in the example above, the type of a needs to verify the following equation

$$\Gamma(a) = bool \xrightarrow{\Gamma(a)} t$$

for some t. This is not possible, hence this expression cannot be typed.

We describe a type system that ensures the termination of expressions typed with a secret pc by disallowing fixpoints and their encoding through references. This type system also enforces the annotations (L/H) on conditional branches depending on the level of the guard so that we know if we need to use the alternate semantic rules or if we can safely use the original ones.

Here are our updated security types:

$$\begin{array}{rcl}
\Upsilon & ::= & \widetilde{\tau} \mid L \\
t & ::= & bool \mid \tau\ \mathsf{ref} \mid \tau \xrightarrow{\Upsilon} \tau \\
\tau & ::= & t^{\ell}
\end{array}$$

We introduce a new type of pc (Υ) that is either L or a set of security types. We let $\Upsilon \leq \Upsilon'$ if $\Upsilon = L$ or $\Upsilon' \neq L$ and $\Upsilon' \subseteq \Upsilon$ (and we define the associated least upper bound and greatest lower bound accordingly). In the updated security type for abstractions, the label is either L if the pc of the function is low (we don't need to restrict functions with a low pc since they cannot appear in H-conditionals) or the set of the types of the references it accesses. In the following, we consider environments and memory typing with these new types.

We now equip our language with a type system that enforces (together with the extended semantics rules) path non-interference and termination for expressions with a secret pc. Typing judgments for values are similar to those of the original type system. Typing judgments for expressions are of the form $\Upsilon, \Gamma \vdash e : \tau$, with our new pc ($\Upsilon$). The typing rules appear in Figure 6. Most typing rules for values and expressions are similar to those of the type system for IO non-interference. In the typing rule for recursive functions, we now require all recursive calls to be run at a low pc (since those can cause non-termination). In the typing rule for dereference, we add the reference accessed to the pc, thus the pc of an expression is either L or the set of the references it access.

We first show that this type system does enforce termination for expressions that type with a pc that is not low.

Theorem 6: Let Γ be an environment, let μ be a typed memory, let e be an expression, let τ be a security type,

Relations between types

$$\frac{\ell \leq \ell' \qquad t \leq t'}{t^{\ell} \leq t'^{\ell'}} \qquad \frac{\tau_0' \leq \tau_0 \qquad \tau_1 \leq \tau_1' \qquad \Upsilon' \leq \Upsilon}{(\tau_0 \xrightarrow{\Upsilon} \tau_1) \leq (\tau_0' \xrightarrow{\Upsilon'} \tau_1')}$$

Type judgments for values

$$\Gamma \vdash k : bool^{L} \qquad \Gamma \vdash \bot : \tau \qquad \frac{\Gamma \vdash v : \tau \qquad \tau \leq \tau'}{\Gamma \vdash v : \tau'}$$

$$\frac{\Gamma(a) = \tau}{\Gamma \vdash a : \tau\ \mathsf{ref}^{L}} \qquad \Gamma \vdash x : \Gamma(x) \qquad \frac{\Upsilon, \Gamma; x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x.e : (\tau \xrightarrow{\Upsilon} \tau')^{L}}$$

$$\frac{\Upsilon, \Gamma; x : \tau; f : (\tau \xrightarrow{L} \tau')^{L} \vdash e : \tau'}{\Gamma \vdash \mathsf{fix}\ f.\lambda x.e : (\tau \xrightarrow{\Upsilon} \tau')^{L}}$$

Type judgments for expressions

$$\frac{\Upsilon, \Gamma \vdash e : t^{\ell} \qquad \tau \leq \tau' \qquad \Upsilon' \leq \Upsilon \qquad t \leq t'}{\Upsilon', \Gamma \vdash e : \tau'}$$

$$\frac{\Gamma \vdash v : \tau}{\emptyset, \Gamma \vdash v : \tau} \qquad \frac{\Gamma \vdash v : bool^{\ell} \qquad \Gamma \vdash v' : bool^{\ell}}{\emptyset, \Gamma \vdash v \wedge v' : bool^{\ell}}$$

$$\frac{\Gamma \vdash v : bool^{\ell} \qquad \Gamma \vdash v' : bool^{\ell}}{\emptyset, \Gamma \vdash v \vee v' : bool^{\ell}} \qquad \frac{\Gamma \vdash v : bool^{\ell}}{\emptyset, \Gamma \vdash \neg v : bool^{\ell}}$$

$$\frac{\Upsilon, \Gamma \vdash e : (\tau' \xrightarrow{\Upsilon'} \tau)^{\ell} \qquad \Upsilon, \Gamma \vdash e' : \tau' \qquad \Upsilon' = L \Rightarrow \ell = L}{\Upsilon \sqcap \Upsilon', \Gamma \vdash e\ e' : \tau \sqcup \ell}$$

$$\frac{\Gamma \vdash v : \mathsf{bool}^{\ell} \qquad \Upsilon, \Gamma \vdash e_1 : \tau \qquad \Upsilon = L \Rightarrow \ell = L \qquad \Upsilon, \Gamma \vdash e_0 : \tau}{\Upsilon, \Gamma \vdash \mathsf{If}_{\ell}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0 : \tau \sqcup \ell}$$

$$\frac{\Upsilon, \Gamma \vdash e : T^{\ell} \qquad \Upsilon = L \Rightarrow \ell = L}{\Upsilon, \Gamma \vdash \mathsf{ref}\ e : T^{\ell}\ \mathsf{ref}^{L}}$$

$$\frac{\Upsilon, \Gamma \vdash e : T^{\ell}\ \mathsf{ref}^{\ell'}}{\Upsilon \cup \{T^{\ell'}\}, \Gamma \vdash !e : T^{\ell \sqcup \ell'}}$$

$$\frac{\Upsilon, \Gamma \vdash v : T^{\ell}\ \mathsf{ref}^{\ell} \qquad \Upsilon, \Gamma \vdash v' : T^{\ell} \qquad \Upsilon = L \Rightarrow \ell = L}{\Upsilon, \Gamma \vdash v := v' : bool^{L}}$$

Figure 6. Type system for termination and path non-interference

and let Υ be a pc such that $\Upsilon \neq L$ and $\Upsilon, \Gamma \vdash e : \tau$. There exists no infinite sequence of configurations $\langle e_i, \mu_i \rangle$ and infinite sequence α_i such that

$$\langle e, \mu \rangle \xrightarrow{\alpha_0} \langle e_1, \mu_1 \rangle \xrightarrow{\alpha_1} \dots$$

We now show that if an expression terminates with the standard rules, and can be typed in this type system, then it terminates when we use the additional rules for H-conditionals. This means, in particular that typed expressions terminate as often with the additional rules.

Theorem 7: Let Γ be an environment, let μ be a typed memory, let e be an expression, let τ be a security type, and let Υ be a pc such that $\Upsilon, \Gamma \vdash e : \tau$. If there exists a sequence of configurations $\langle e_i, \mu_i \rangle_{i \leq n}$, and a sequence $(\alpha_i)_{i \leq n}$ such that

$$\langle e, \mu \rangle \xrightarrow{\alpha_0} \langle e_1, \mu_1 \rangle \xrightarrow{\alpha_1} \dots \langle e_n, \mu_n \rangle$$

in the original rules, and where e_n is irreducible, then there exists no infinite sequence of configurations $\langle e_i', \mu_i' \rangle_i$ and infinite sequence $(\alpha_i')_i$ such that

$$\langle e, \mu \rangle \xrightarrow{\alpha_0'} \langle e_1', \mu_1' \rangle \xrightarrow{\alpha_1'} \dots$$

Last, and this is our main theorem, we show that typed expressions are path non-interferent with the additional rules.

Theorem 8: Let Γ be an environment, let e be an expression, let τ be a security type, and let Υ be a pc such that $\Upsilon, \Gamma \vdash e : \tau$. Then e is path non-interferent. when executed with the additional rules.

VI. Timing Channel

In the section above, we offered a semantics solution for control-flow leaks. Incidentally, since the control-flow of programs does not depend on the private data anymore, the code executed does not depend on the private data either, hence our solution also closes timing channels.

In order to show that no part of the executed code depends on the private data, we now annotate every semantic rule with its own label (we omit similar labels for the extended rules):

$$\begin{array}{rcll}
\langle k \wedge k', \mu \rangle & \xrightarrow{\wedge} & \langle k'', \mu \rangle & \text{where } k'' = k \wedge k' \\
\langle k \vee k', \mu \rangle & \xrightarrow{\vee} & \langle k'', \mu \rangle & \text{where } k'' = k \vee k' \\
\langle \neg k, \mu \rangle & \xrightarrow{\neg} & \langle k', \mu \rangle & \text{where } k' = \neg k \\
\langle (\lambda x.e)v, \mu \rangle & \xrightarrow{\lambda} & \langle e[v/x], \mu \rangle & \\
\langle (\mathsf{fix}\ f.\lambda x.e)v, \mu \rangle & \xrightarrow{\mathsf{fix}} & \langle e[v/x][(\mathsf{fix}\ f.\lambda x.e)/f], \mu \rangle & \\
\langle \mathsf{If}_L\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0, \mu \rangle & \xrightarrow{1} & \langle e_1, \mu \rangle & \text{when } v = 1 \\
\langle \mathsf{If}_L\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0, \mu \rangle & \xrightarrow{0} & \langle e_0, \mu \rangle & \text{when } v = 0 \\
\langle \mathsf{ref}\ v, \mu \rangle & \xrightarrow{\nu a} & \langle a, \mu\{a \mapsto v\} \rangle & \text{for } a \notin \mathsf{dom}(\mu) \\
\langle !v, \mu \rangle & \xrightarrow{!v} & \langle \mu(v), \mu \rangle & \text{when } v \in \mathsf{dom}(\mu) \\
\langle v := v', \mu \rangle & \xrightarrow{v:=} & \langle 0, \mu\{v \mapsto v'\} \rangle & \text{when } v \in \mathsf{dom}(\mu) \\
 & \dots & &
\end{array}$$

These annotations on the behaviors of programs are very precise; still, we can show that typed programs stay non-interferent.

Time-sensitive non-interference: Similarly to path non-interference, an expression is considered secure when the initial values of the privates variables do not influence the final values of the public variables or the semantics annotations. However, in this case, the semantic annotations carry more information. For instance, the length of the program execution, or even the commands executed or the references accessed are observable through the annotations.

Definition 2: Let Γ be an environment, ℓ a label, and e an expression. We say that e is time non-interferent when, for every memories μ_0 and μ_1 that contain only booleans and such that $\mu_0 =_L \mu_1$, and for every memory μ_0', value v_0, and trace γ_0 such that

$$\langle e, \mu_0 \rangle \xrightarrow{\gamma_0}{}^{*} \langle v_0, \mu_0' \rangle$$

there exist a memory μ_1', a value v_1, and a trace γ_1 such that

$$\langle e, \mu_1 \rangle \xrightarrow{\gamma_1}{}^{*} \langle v_1, \mu_1' \rangle$$

$\mu_0' =_L \mu_1'$, and $\gamma_0 = \gamma_1$.

We now restate our theorem:

Theorem 9: Let Γ be an environment, let e be an expression, let t^ℓ be a security type, and let Υ be a pc such that $\Upsilon, \Gamma \vdash e : t^\ell$. Then e is time non-interferent when executed with the extended rules.

VII. Related work

The work described in this paper is part of a larger project exploring computation on encrypted data. A previous paper from our group Mitchell et al. [2012] describes an implementation of oblivious computations on a simpler and more restricted language with related semantics, using either homomorphic encryption or secure multiparty computation for secrecy. The more recent compiler developed by our group takes source code corresponding to the present paper as input, typechecks it, then produces Haskell code that implements the semantics of this paper using optimized libraries (from another team collaborating with us) to execute the cryptographic operations. The implementation ensures strong correctness and security properties, subject to some assumptions on the cryptographic operations.

Our extended semantics is used in the implementation of oblivious computation because most operations can be executed without reading secret values. The only operations that need to read those values are boolean operations, since we never directly reduce a conditional with a confidential guard. It is possible to implement these operations using, for instance, homomorphic encryption or secure multiparty computation. With fully homomorphic encryption, we can execute our boolean operations on encrypted values without revealing them, so we can implement our semantics with encrypted blobs for secret values. In the secure multiparty computation case, the secrets are shared between a number of servers. The servers can compute on the original secret by computing on their share and exchanging some information. If enough servers do not collude, none of them learn anything about the secrets. Once again, we can implement our semantics using those shares for secret values.

A large part of the slowdown (presently potentially on the order of 10^9 for homomorphic encryption) currently comes from the cryptographic operations, but some of it comes from the principle discussed in this paper that we need oblivious executions. In related work, our team has worked on faster oblivious algorithms for specific computational problems of interest. While homomorphic encryption may be prohibitively slow for some applications, the same programs run with practical running time when executed using secret sharing for confidentiality.

Other systems for secure computations include SCET [Bogetoft et al., 2005], with focus on economic applications and secure double auctions; FairplayMP [Ben-David et al., 2008], a specification language SFDL that is converted to primitive operations on bits; Sharemind [Bogdanov et al., 2008], for multiparty computations on large datasets; VIFF [Damgård et al., 2009], a basic language embedded in Python and API to cryptographic primitives. These systems implement cryptographic protocols, without proving the more comprehensive correctness and security properties. Similarly, SMCL [Nielsen and Schwartzbach, 2007] is an imperative-style DSL implemented with secure multiparty computation, that has some correctness and security properties.

VIII. Conclusion

We have studied path-based information-flow using a lambda calculus with references whose semantics represents information leaked to an adversary that can see the execution path. Starting with a type system (similar to the one of flow-caml [Pottier and Simonet, 2003]), we establish baseline soundness for IO non-interference and consider the additional path information-flow problem. One alternative for preventing path-based leakage is our first extended information-flow type system (Theorem 2); equivalently we present a program transformation making path leakage explicit, reducing path information-flow to IO information-flow (Theorem 3).

It is relatively easy to see that simply preventing path information leakage is very restrictive – no program can branch according to any confidential value. We can provide greater expressiveness using an alternate semantics that effectively executes multiple paths and combines the results using data operations. This semantics is implementable using homomorphic encryption [Gentry, 2009, Gentry, 2010, van Dijk et al., 2010] or secure multiparty computation [Ben-Or et al., 1988, Gennaro et al., 1998, Cramer et al., 2000, Naor and Nissim, 2001] because such data operations hide the data values from the execution platform. We prove

that this semantics has a weak confluence property we call *convergence,* implying that the semantics of a program is well-defined and equivalent in output to its standard semantics, with fewer path-flow leaks (Theorem 4). Because executing all paths can force nontermination when loop iteration depends on a secret value, we extend our security type system to prevent loops that necessarily do not terminate for this reason. We prove that well-typed programs under this discipline terminate in the new semantics whenever they would terminate under the standard semantics (Theorem 6) and prove path-flow non-interference (Theorem 7). Thus we have developed and proved sound a language and type system that we believe is suitable for programming in an environment where path information must be hidden from an adversary. This language is more cleanly defined, more flexible for programming (for instance, we allow functions in references), and has cleaner and more comprehensive formal analysis than previous efforts (see [Bain et al., 2011, Mitchell et al., 2012]).

In order to put our main results in perspective, we investigate the connection between path-flow information leakage and timing channels. We formalize timing leakage and prove that our system preventing path-flow leakage also prevents timing leakage (Theorem 8). Intuitively, this is to be expected when data operations do not leak data values, because timing difference arise from the differences between execution paths.

REFERENCES

J. Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 40–53. ACM, 2000.

A. Bain, J. C. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman. A domain-specific language for computing on encrypted data (invited talk). In *FSTTCS*, pages 6–24, 2011.

A. Ben-David, N. Nisan, and B. Pinkas. Fairplaymp: a system for secure multi-party computation. In *ACM Conference on Computer and Communications Security*, pages 257–266, 2008.

M. Ben-Or, S. Goldwasser, and A. Wigderson. Completeness theorems for noncryptographic fault-tolerant distributed computations. In *Proc. of STOC*, pages 1–10, 1988.

D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008.

P. Bogetoft, I. B. Damgard, T. Jakobsen, K. Nielsen, J. Pagter, and T. Toft. Secure computing, economy, and trust: A generic solution for secure auctions with real-world applications, 2005.

R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, CCSW '09, pages 85–90, 2009. ISBN 978-1-60558-784-4.

R. Cramer, I. Damgård, and U. M. Maurer. General secure multi-party computation from any linear secret-sharing scheme. In *EUROCRYPT*, pages 316–334, 2000.

I. Damgård, M. Geisler, M. Krøigaard, and J. B. Nielsen. Asynchronous multiparty computation: Theory and implementation. In *Public Key Cryptography*, pages 160–179, 2009.

R. Gennaro, M. O. Rabin, and T. Rabin. Simplified VSS and fast-track multiparty computations with applications to threshold cryptography. In *Proc. ACM Symp. Principles of Distributed Computing (PODC)*, pages 101–111, 1998.

C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09: Proc. ACM Symposium on Theory of Computing*, pages 169–178, 2009.

C. Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53(3):97–105, 2010.

J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

IDC. IT Cloud Services User Survey, pt.2: Top Benefits & Challenges, 2008. `http://blogs.idc.com/ie/?p=210`, accessed Nov. 2011.

J. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

J. C. Mitchell, R. Sharma, D. Stefan, and J. Zimmerman. Information-flow control for programming on encrypted data. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 45–60. IEEE, June 2012.

D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology-ICISC 2005*, pages 156–168. Springer, 2006.

M. Naor and K. Nissim. Communication preserving protocols for secure function evaluation. In *Proc. of STOC'01*, pages 590–599, 2001.

J. Nielsen and M. Schwartzbach. A domain-specific programming language for secure multiparty computation. In *PLAS '07: Programming languages and analysis for security*, pages 21–30. ACM, 2007.

B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

F. Pottier and V. Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, Jan. 2003. ©ACM.

A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Proceedings of the*

25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 355–364. ACM, 1998.

M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan. Fully homomorphic encryption over the integers. In *Proc. of Eurocrypt'10*, pages 24–43, 2010.

D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of computer security*, 4 (2/3):167–188, 1996.

APPENDIX: PROOF SKETCHES

This appendix contains proof sketches for the most interresting theorem, that is Theorems 4,6-9.

A. Proof of Theorem 4

Theorem 4 is about convergence of our original and extended rules together. In the extended rules, the evaluation of conditionals can be delayed. Hence we introduce an equivalence between expressions that contain additional conditionals.

Definition 3: We let $\equiv$ be the symmetric relation on expressions and values defined by:

$$e \equiv e \qquad \lambda x.e_k \equiv \lambda x.\mathsf{If}\ k\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0$$

$$\mathsf{fix}\ f.\lambda x.e_k \equiv \mathsf{fix}\ f.\lambda x.\mathsf{If}\ k\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0$$

$$\frac{e_1 \equiv e_1' \qquad e_0 \equiv e_0'}{\mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0 \equiv \mathsf{If}\ v\ \mathsf{then}\ e_1'\ \mathsf{else}\ e_0'}$$

$$\frac{e_0 \equiv e_0' \qquad e_1 \equiv e_1'}{e_0\ e_1 \equiv e_0'\ e_1'} \qquad \frac{e \equiv e'}{\mathsf{ref}\ e \equiv \mathsf{ref}\ e'} \qquad \frac{e \equiv e'}{!e \equiv !e'}$$

$$\frac{e_0 \equiv e_0' \qquad e_1 \equiv e_1'}{e_0 := e_1 \equiv e_0' := e_1'} \qquad \frac{e \equiv e'}{\lambda x.e \equiv \lambda x.e'}$$

$$\frac{e \equiv e'}{\mathsf{fix}\ f.\lambda x.e \equiv \mathsf{fix}\ f.\lambda x.e'}$$

(We omit the symmetric rules.)

This lemma is a more generic version of Theorem 4 using our equivalence.

Lemma 1: Let μ_0 and μ_1 be two pure memories such that, for every location l, $\mu_0(l) \equiv \mu_1(l)$. Let e_0 and e_1 be two expressions such that $e_0 \equiv e_1$. If $\langle e_0, \mu_0\rangle \xrightarrow{\alpha_0} \langle e_0', \mu_0'\rangle$ and $\langle e_1, \mu_1\rangle \xrightarrow{\alpha_1} \langle e_1', \mu_1'\rangle$ then, there exists e_0'', e_1'', μ_0'', μ_1'', γ_0', and γ_1' such that $\langle e_0', \mu_0'\rangle \xrightarrow{\gamma_0'}{}^{*} \langle e_0'', \mu_0''\rangle$, $\langle e_1', \mu_1'\rangle \xrightarrow{\gamma_1'}{}^{*} \langle e_1'', \mu_1''\rangle$, $e_0'' \equiv e_1''$, and, for every location l, $\mu_0'(l) \equiv \mu_1'(l)$.

Proof of Lemma 1: Let μ_0 and μ_1 be two pure memories such that, for every location l, $\mu_0(l) \equiv \mu_1(l)$. Let e_0 and e_1 be two expressions such that $e_0 \equiv e_1$. We first prove the property on pure expressions, then on all expressions. Thus, we can apply Lemma 1 on the pure expressions stored in memory when doing the proof. The proof is by induction on the size of the expressions. We omit the proof for pure expressions since its cases are a subset of the general proof. We now sketch the examination of all possible couple of reductions.

- If both reductions are memory reductions:

$$\frac{\langle e^0, \mu\rangle \xrightarrow{\alpha} \langle e^{0\prime}, \mu\rangle \qquad \mu(l_0) = e^0}{\langle e_0, \mu\rangle \xrightarrow{\alpha} \langle e_0, \mu[l_0 \mapsto e^{0\prime}]\rangle}$$

$$\frac{\langle e^1, \mu\rangle \xrightarrow{\alpha} \langle e^{1\prime}, \mu\rangle \qquad \mu(l_1) = e^1}{\langle e_1, \mu\rangle \xrightarrow{\alpha} \langle e_1, \mu[l_1 \mapsto e^{1\prime}]\rangle}$$

  If $l_0 = l_1$, we conclude by Lemma 1 on pure expressions e^0 and e^1, otherwise, we reduce l_1 in $\langle e_0, \mu[l_0 \mapsto e^{0\prime}]\rangle$ and conclude.
- If one of the reduction is a reduction of
  - $v \wedge v'$,
  - $v \vee v'$,
  - $\neg v$,
  - $\lambda x.e\ v$,
  - $\mathsf{fix}\ f.\lambda x.e\ v$,
  - $\mathsf{ref}\ v$,
  - $!v$,
  - or $v := v'$,

  and the other reduction is a memory reduction, then we can carefully mirror the reductions and conclude (we may have to add or remove one reduction in $!v$, $v := v'$, $\lambda x.e\ v$, and $\mathsf{fix}\ f.\lambda x.e\ v$ cases). If the other reduction is similar, the resulting expressions are already equivalent (except in cases $\lambda x.e\ v$ and $\mathsf{fix}\ f.\lambda x.e\ v$ where we may need to add a step.
- There are seven possible type of reductions (numbered $1-7$ in the following) from expressions of the form $\mathsf{If}\ k\ \mathsf{then}\ e^1\ \mathsf{else}\ e^0$, (we assume without loss of

generality that $k = 1$):

$$\langle \mathsf{If}\ 1\ \mathsf{then}\ e^1\ \mathsf{else}\ e^0, \mu\rangle \xrightarrow{1} \langle e^1, \mu\rangle$$

$$\langle \mathsf{If}\ 1\ \mathsf{then}\ k_1\ \mathsf{else}\ k_0, \mu\rangle \rightarrow \langle (1 \wedge k_1) \vee (\neg 1 \wedge k_0), \mu\rangle$$

$$\frac{\langle e^1, \mu\rangle \xrightarrow{\alpha} \langle e^{1\prime}, \mu'\rangle \quad \mu''(x) = \mathsf{If}\ 1\ \mathsf{then}\ \mu'(x)\ \mathsf{else}\ \mu(x)\ \text{for}\ x \in dom(\mu')}{\langle \mathsf{If}\ 1\ \mathsf{then}\ e^1\ \mathsf{else}\ e^0, \mu\rangle \xrightarrow{\alpha} \langle \mathsf{If}\ 1\ \mathsf{then}\ e^{1\prime}\ \mathsf{else}\ e^0, \mu''\rangle}$$

$$\frac{\langle e^0, \mu\rangle \xrightarrow{\alpha} \langle e^{0\prime}, \mu'\rangle \quad \mu''(x) = \mathsf{If}\ 1\ \mathsf{then}\ \mu'(x)\ \mathsf{else}\ \mu(x)\ \text{for}\ x \in dom(\mu')}{\langle \mathsf{If}\ 1\ \mathsf{then}\ e^1\ \mathsf{else}\ e^0, \mu\rangle \xrightarrow{\alpha} \langle \mathsf{If}\ 1\ \mathsf{then}\ e^1\ \mathsf{else}\ e^{0\prime}, \mu''\rangle}$$

$$\langle \mathsf{If}\ 1\ \mathsf{then}\ \lambda x.e^{1\prime}\ \mathsf{else}\ \lambda x.e^{0\prime}, \mu\rangle \rightarrow \langle \lambda x.\mathsf{If}\ 1\ \mathsf{then}\ e^{1\prime}\ \mathsf{else}\ e^{0\prime}, \mu\rangle$$

$$\langle \mathsf{If}\ 1\ \mathsf{then}\ \mathsf{fix}\ f.\lambda x.e^{1\prime}\ \mathsf{else}\ \mathsf{fix}\ f.\lambda x.e^{0\prime}, \mu\rangle \rightarrow \langle \mathsf{fix}\ f.\lambda x.\mathsf{If}\ 1\ \mathsf{then}\ e^{1\prime}\ \mathsf{else}\ e^{0\prime}, \mu\rangle$$

$$\frac{\langle e', \mu\rangle \xrightarrow{\alpha} \langle e'', \mu\rangle \qquad \mu(l) = e'}{\langle \mathsf{If}\ 1\ \mathsf{then}\ e^1\ \mathsf{else}\ e^0, \mu\rangle \xrightarrow{\alpha} \langle \mathsf{If}\ 1\ \mathsf{then}\ e^1\ \mathsf{else}\ e^2, \mu[l \mapsto e'']\rangle}$$

We can verify that

- Reduction 1 when followed by a reduction on e^1, gives the same result than reduction 3 followed by 1 and reductions on memory. If $e_0 = e_1$, we conclude for these reductions, otherwise, we proceed by induction on the first branches.
- Reduction 1 gives the same result than reduction 4 followed by 1 and reductions on memory.
- Reduction 1 gives the same result than reduction 2 followed by three reductions on $\wedge$ and $\vee$.
- 1 and 5 (or 6) give equivalent expressions.
- 7 easily commutes with 1.

- If both reductions are λ applications (and similarly for fixpoints):
  - If both e_0 and e_1 are of the form $(\lambda x.e)\ e'$, we conclude.
  - If e_0 is of the form $(\lambda x.e)\ e'$ and e_1 is of the form $\lambda x.\mathsf{If}\ k\ \mathsf{then}\ e^1\ \mathsf{else}\ e^0\ e'$ (where e' is a value or a tree of references and $e_k \equiv e$), we can mirror one step of e_0 with two steps of e_1.
- If the two reductions are contextual, we conclude by induction.
- If one of the reduction is of the form $\langle !\mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, \mu\rangle \rightarrow \langle \mathsf{If}\ v\ \mathsf{then}\ !e_1\ \mathsf{else}\ !e_2, \mu\rangle$

  $\langle (\mathsf{If}\ v\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) := e_3, \mu\rangle \rightarrow \langle \mathsf{If}\ v\ \mathsf{then}\ e_1 := e_3\ \mathsf{else}\ e_2 := e_3, \mu\rangle$

  $\langle v := (\mathsf{If}\ v'\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1), \mu\rangle \rightarrow \langle \mathsf{If}\ v'\ \mathsf{then}\ v := e_0\ \mathsf{else}\ v := e_1, \mu\rangle$

  $\langle \mathsf{ref}\ (\mathsf{If}\ v'\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1), \mu\rangle \rightarrow \langle \mathsf{If}\ v'\ \mathsf{then}\ \mathsf{ref}\ e_0\ \mathsf{else}\ \mathsf{ref}\ e_1, \mu\rangle$

  and the other is contextual, we verify that we can converge by reducing the conditional.

■

Proof: The proof of Theorem 4 follows from Lemma 1. ■

B. Proof of Theorem 6

This proof of termination is similar to the proof of termination for simply typed lambda calculus, but references add complexity. We first inductively define a notion of *reducible expressions* for a particular set of types. This notion implies termination. Then, we show some properties of this definition (conservation by reduction, and independence from the set of types). Finally, we prove that all typed expression with a high pc are reducible.

Definition 4: Let Υ be a set of types. We define *reducible expressions* for this set inductively. Let e be an expression such that $\Upsilon, \Gamma \vdash e : \tau$. Let $(l_i)_{i \leq n}$ be the set of locations in Γ such that $\Gamma(l_i) \in \Upsilon, l \leq n$, e is reducible for Υ if:

- τ is atomic and for every series of pure expressions $(e_1^i, \ldots, e_n^i)_i$ such that $\emptyset, \Gamma \vdash e_j^i : \Gamma(l_j), j \leq n$ and e_j^i is reducible for $\Upsilon \backslash \Gamma(l_j)$, there exists no infinite series of states $(\langle e_i, \mu_i\rangle)_i$ such that $e_0 = e$, and $\langle e_i, l_j \mapsto e_j^i, j \leq n\rangle \rightarrow \langle e_{i+1}, \mu_{i+1}\rangle$; and for every such finite chain, the $(\mu_i)_i$ contain expressions that are reducible for $\Upsilon \backslash \Gamma(l)$ (and with $\Upsilon = \emptyset$).
- τ is of the form $(\tau' \xrightarrow{\Upsilon'} \tau'')^\ell$ with $\Upsilon' \subseteq \Upsilon$ and for every expression e' reducible for Υ and such that $\Upsilon, \Gamma \vdash e' : \tau$, we have $e\ e'$ reducible for Υ.

This induction is well founded since we will only make inductive calls on τ', Υ' such that $\Upsilon' \subset \Upsilon$ or $\Upsilon' \subseteq \Upsilon$ and $\tau' \in \tau$.

In this lemma, we prove that the notion of reducibility is conserved by reduction.

Lemma 2: If e is reducible for Υ, and $\langle e, \mu\rangle \rightarrow \langle e', \mu'\rangle$ for some μ that verifies the conditions of reducibility, then e' is reducible for Υ.

Proof of Lemma 2: The proof is by definition of reducibility. ■

In this lemma, we prove that a conditional is reducible if and only if its branches are reducible.

Lemma 3: $\mathsf{If}_H\ v\ \mathsf{then}\ e_0\ \mathsf{else}\ e_1$ is reducible if and only if e_0 and e_1 are reducible.

Proof of Lemma 3: If one step is made by e_0 or e_1, we conclude by induction (on the number of steps for them to terminate). The other cases are also by induction. ∎

In this lemma, we prove that if an expression is reducible for a particular set of types, it is reducible for smaller sets of types.

Lemma 4: Let e be an expression reducible for Υ, τ a type and $\Upsilon' \subseteq \Upsilon$ such that $\Upsilon', \Gamma \vdash e : \tau$ and, for every $\Upsilon'' \in \tau$, $\Upsilon'' \subseteq \Upsilon'$. Then e is reducible for Υ'.

Proof of Lemma 4: The proof is by induction on the definition of reducible

- If τ is atomic, the series of pure expressions $(e_1^i, \ldots, e_n^i)_i$ such that $\emptyset, \Gamma \vdash e_j^i : \Gamma(l_j), j \leq n$ and e_j^i is reducible for $\Upsilon' \backslash \Gamma(l_j)$ are the same than for $\Upsilon \backslash \Gamma(l_j)$ by induction.
- Otherwise, we conclude similarly by induction.

∎

In this lemma, we prove that if an expression is reducible for a particular set of types, it is reducible for bigger sets of types.

Lemma 5: If e is reducible for Υ, $\Upsilon \subseteq \Upsilon'$, then e is reducible for Υ'.

Proof of Lemma 5: The proof is by induction on the definition of reducible

- If τ is atomic, the series of pure expressions $(e_1^i, \ldots, e_n^i)_i$ such that $\emptyset, \Gamma \vdash e_j^i : \Gamma(l_j), j \leq n$ and e_j^i is reducible for $\Upsilon' \backslash \Gamma(l_j)$ are the same than for $\Upsilon \backslash \Gamma(l_j)$ by induction.
- Otherwise, lets consider an argument that corresponds to the definition; after a bunch of reductions, we get $\langle v\ e', \mu' \rangle$ where e' is a value or a tree of locations. At this point, e' types with $\emptyset$, and we can conclude with the Lemma 4 that it is reducible for Υ.

∎

Lemma 6: Every typed expression e is reducible.

Proof of Lemma 6: We show by induction on the structure of e that if $\Upsilon, \Gamma \vdash e : \tau$ for some Υ, Γ, τ and $e_1, \ldots, e_n$ are reducible pure expressions with $\emptyset, \Gamma \vdash e_i : \Gamma(x_i)$, then the term $e[e_1/x_1, \ldots, e_n/x_n]$ is reducible.

- If e is a variable, $e[e_1/x_1, \ldots, e_n/x_n]$ is a variable or one of the e_i, we conclude.
- If $e = e'\ e''$, we conclude by induction.
- If $e = \lambda x.e'$, we have $e[e_1/x_1, \ldots, e_n/x_n] = \lambda x.e'[e_1/x_1, \ldots, e_n/x_n]$. We let $e'' = e'[e_1/x_1, \ldots, e_n/x_n]$. Let $\tau_0 \xrightarrow{\Upsilon_0} \tau_1 \xrightarrow{\Upsilon_1} \ldots \xrightarrow{\Upsilon_{m-1}} \tau_m$ be the type of $\lambda x.e''$, and let $e^0, e^1, \ldots, e^m$ be reducible terms of types $\tau_0, \tau_1, \ldots, \tau_n$. Let μ be a memory satisfying the properties of the reducible definition. Lets consider chain of reductions of the state $\langle \lambda x.e''\ e^0\ e^1\ \ldots\ e^m, \mu \rangle$. In this chain, either the first application is never reduced (and we conclude), or it is reduced at step k. In this case, the term after step k is of the form: $\langle e''[e^{0*}/x]\ e^1\ \ldots\ e^m, \mu' \rangle$, where e^{0*} is a reduction of e^0 and either a value or a tree of locations (hence reducible for Υ), we conclude by induction on e''.
- If $e = !e'$, we proceed similarly. Either ! is never reduced, (and we conclude), or the term after the reduction step is of the form:
  - $\langle (\mathsf{If}_H\ v\ \mathsf{then}\ !e^{0*0}\ \mathsf{else}\ !e^{0*1})\ e^1\ \ldots\ e^m, \mu' \rangle$ if $\mathsf{If}\ v\ \mathsf{then}\ e^{0*0}\ \mathsf{else}\ e^{0*1}$ is a reduction of e, in which case we use Lemma 3 to separate the two branches and continue reductions; or
  - $\langle e''\ e^1\ \ldots\ e^m, \mu' \rangle$, where $e'' = \mu'(l)$ and l is a reduction of e^0. e'' is reducible for $\Upsilon \backslash \tau''$, but it is also reducible by Υ from Lemma 5.
- If $e = \mathsf{If}\ e'\ \mathsf{then}\ e''\ \mathsf{else}\ e'''$ we use Lemma 3 and conclude by induction.
- If $e = \mathsf{ref}\ e'$, we conclude similarly by induction.
- If $e = (e' := e'')$, we conclude similarly by induction.

∎

Proof: The proof of Theorem 6 follows from Lemma 6. ∎

C. Proof of Theorem 7

This theorem is about termination in two different reduction system (with the original or extended rules). In order to relate executions, and since extended rules delay the executions of conditionals, we introduce a relation from expressions to expressions that contain additional conditionals.

Definition 5: We let $\preceq$ be a transitive relation on expressions and values defined by

$$e \preceq e \qquad e_k \preceq \mathsf{If}_H\ k\ \mathsf{then}\ e_1\ \mathsf{else}\ e_0$$

$$v_k \preceq (k \wedge v_1) \vee (\neg k \wedge v_0) \qquad v_0 \preceq 0 \vee (\neg 0 \wedge v_0)$$

$$v_0 \preceq (\neg 0 \wedge v_0) \qquad v_0 \preceq (1 \wedge v_0) \qquad \frac{e_0 \preceq e_0' \qquad e_1 \preceq e_1'}{e_0\ e_1 \preceq e_0'\ e_1'}$$

$$\ldots$$

(We omit the similar rules for $k = 1$ and most inductive rules.)

In this lemma, we prove that if a value is in relation with an expression (by $\preceq$), then every intermediate state is in relation with the original expression.

Lemma 7: Let Γ be an environment, let v_0 be a typed value, and e_1 be a typed expression such that $v_0 \preceq e_1$, let μ_0 be a value memory and μ_1 be a pure memory such that, for every location l, $\mu_0(l) \preceq \mu_1(l)$. Let $\langle e^i, \mu^i \rangle_{i \leq k}$ be a sequence of configuration terminating on an irreducible expression, and $(\alpha^i)_{i \leq k}$ be a sequence of transition labels such that

$$\langle e_1, \mu_1 \rangle = \langle e^1, \mu^1 \rangle \xrightarrow{\alpha^1} \ldots \langle e^k, \mu^k \rangle$$

For every $i \leq k$, $v_0 \preceq e^i$, and $\mu_0(l) \preceq \mu^i(l)$.

Proof of Lemma 7: The proof is by induction on the length of the trace (k), with a first pass on pure expressions so that we can apply the lemma on memory reductions.

- If the next reduction is a memory reduction, we conclude by induction.
- If e_1 is a value, we conclude.
- If e_1 is of the form

$$(0 \wedge v_1) \vee (\neg 0 \wedge v_0) \qquad (0 \wedge v_1) \vee (\neg 0 \wedge v_0)$$

$$0 \vee (\neg 0 \wedge v_0) \qquad (\neg 0 \wedge v_0) \qquad (1 \wedge v_0)$$

  we reduce and conclude.
- If e_1 is of the form $\mathsf{If}_H\ k\ \mathsf{then}\ e_1^1\ \mathsf{else}\ e_1^0$ and $v_0 \preceq e_1^k$, there are three possible cases:
  - e_1^1 or e_1^0 reduces in the conditional, in which case we conclude by induction.
  - e_1^1 and e_1^0 are values, and the conditional reduces with one of the possible rules for conditionals on values. We verify that the results follows our definition (using induction in one case) and conclude.
  - e_1 is a tree of location, in which case it does not reduces, and we conclude.
- In all other cases where $v_0 \preceq e_1$, e_1 is already a value.

■

In this lemma, we prove that if two expressions are in relation (by $\preceq$), and the second expression terminates, then a reduction of the first expression is eventually mirrored in the second expression.

Lemma 8: Let Γ be an environment, let e_0 and e_1 be two typed expressions such that $e_0 \preceq e_1$, let μ_0 be a value memory, and μ_1 a pure memories such that, for every location l, $\mu_0(l) \preceq \mu_1(l)$. Let $\langle e^i, \mu^i \rangle_{i \leq n}$ be a sequence of configurations, terminating on an irreducible expression, and $(\alpha^i)_{i \leq n}$ be a (possibly infinite) sequence of transition labels such that

$$\langle e_1, \mu_1 \rangle = \langle e^1, \mu^1 \rangle \xrightarrow{\alpha^1} \dots \langle e^n, \mu^n \rangle$$

If $\langle e_0, \mu_0 \rangle \xrightarrow{\alpha} \langle e_0', \mu_0' \rangle$ (in the original semantics), then, there exists an index i such that

- $e_0' \preceq e^i$,
- for every $j < i$, $e_0 \preceq e^j$,
- for every $j < i$, for every location l, $\mu_0(l) \preceq \mu^j(l)$, and
- for every location l, $\mu_0(l) \preceq \mu^n(l)$.

Proof of Lemma 8: The proof is by induction on the length of the trace (k), with a first pass on pure expressions so that we can apply the lemma on memory reductions.

- If the next reduction is a memory reduction, we conclude by induction.
- If e_1 is a value, then e_0 is a value, and we conclude.
- If e_1 is of the form $v \wedge v$, $v \vee v$, or $\neg v$, then e_0 is also of this form (or is a value); we check that the reductions behave correctly and conclude.
- If e_1 is of the form $e_1''\ e_1'''$, then e_0 is of the form $e_0''\ e_0'''$, there are three possibilities:
  - e_0'' is not a value. The only possible reduction is a reduction of e_0''. We conclude by induction on e_0'' and e_1''.
  - e_0'' is a value and e_1'' is not a value, the only possible reductions of e_1 are reductions of e_1''. By Lemma 7 on e_0'' and e_1'' we reduce the problem to the next point.
  - e_0'' and e_1'' are values. e_0''' is not a value; the only possible reduction of e_0 is a reduction of e_0'''. We conclude by induction on e_0''' and e_1'''.
  - e_0'' and e_1'' are values; e_0''' is a value, but e_1''' is not a value or a tree of locations. By Lemma 7 on e_0''' and e_1''', we reduce to the next point.
  - e_0'' and e_1'' are values; e_0''' is a value, e_1''' is a value or a tree of location. The only reduction possible in both cases is the application; we conclude.
- If e_1 is of the form $\mathsf{If}_L\ k\ \mathsf{then}\ e_1^1\ \mathsf{else}\ e_1^0$, then e_0 is of the form $\mathsf{If}_L\ k\ \mathsf{then}\ e_0^1\ \mathsf{else}\ e_0^0$, the only possible reduction of e_0 and e_1 is the standard conditional reduction; we conclude.
- If e_1 is of the form $\mathsf{If}_H\ k\ \mathsf{then}\ e_1^1\ \mathsf{else}\ e_1^0$ and e_0 is of the form $\mathsf{If}_H\ k\ \mathsf{then}\ e_0^1\ \mathsf{else}\ e_0^0$ with $e_0^1 \preceq e_1^1$ and $e_0^0 \preceq e_1^0$, e_0 reduces (with the original rules) to e_0^k, with $e_0^k \preceq \mathsf{If}_H\ k\ \mathsf{then}\ e_1^1\ \mathsf{else}\ e_1^0$ and we conclude.
- If e_1 is of the form $\mathsf{If}_H\ k\ \mathsf{then}\ e_1^1\ \mathsf{else}\ e_1^0$ and $e_0 \preceq e_1^k$, then by induction on e_0 and e_k, there are tow possible cases:
  - e_1^1 or e_1^0 reduces in the conditional, in which case we conclude by induction.
  - e_1^1 and e_1^0 are values or trees of locations, in which cases e_0 is a value (contradicts the hypothesis that e_0 reduces).
- If e_1 is of the form $!\mathsf{If}_H\ k\ \mathsf{then}\ e_1^1\ \mathsf{else}\ e_1^0$ and e_0 is of the form $!\mathsf{If}_H\ k\ \mathsf{then}\ e_0^1\ \mathsf{else}\ e_0^0$ with $e_0^1 \preceq e_1^1$ and $e_0^0 \preceq e_1^0$, e_0 reduces (with the original rules) to $!e_0^k$, with $!e_0^k \preceq !\mathsf{If}_H\ k\ \mathsf{then}\ e_1^1\ \mathsf{else}\ e_1^0$ and we conclude.
- If e_1 is of the form $!\mathsf{If}_H\ k\ \mathsf{then}\ e_1^1\ \mathsf{else}\ e_1^0$ and e_0 is of the form $!e_0^k$ with $e_0^k \preceq e_1^k$, e_1 reduces to $\mathsf{If}_H\ k\ \mathsf{then}\ !e_1^1\ \mathsf{else}\ !e_1^0$ and we conclude by induction.
- In any over case were e_1 is of the form $!e$, we conclude by induction.
- The cases for $\mathsf{ref}\ e$ and $e := e'$ are similar.

■

In this lemma, we generalize the results above for non-terminating expressions.

Lemma 9: Let Γ be an environment, let e_0 and e_1 be two typed expressions such that $e_0 \preceq e_1$, let μ_0 and μ_1 be two pure memories such that, for every location l, $\mu_0(l) \preceq \mu_1(l)$. Let $\langle e^i, \mu^i \rangle_i$ be a sequence of configurations, either infinite or terminating on an irreducible expression, and $(\alpha^i)_i$ be a

(possibly infinite) sequence of transition labels such that

$$\langle e_1, \mu_1\rangle = \langle e^1, \mu^1\rangle \xrightarrow{\alpha^1} \langle e^2, \mu^2\rangle \xrightarrow{\alpha^2} \dots$$

If e_0 is a value, then the sequence is finite, the last expression e^i is such that $e_0 \preceq e^i$, and $\mu_0(l) \preceq \mu^i(l)$. If $\langle e_0, \mu_0\rangle \xrightarrow{\alpha} \langle e'_0, \mu'_0\rangle$ (in the original semantics), then, there exists an index i such that $e'_0 \preceq e^i$, and for every location l, $\mu'_0(l) \preceq \mu^i(l)$.

Proof of Lemma 9: The proof is by induction on the structure of e_1. It is very similar to the proof of Lemma 8 (even if the induction is different), so we only highlight the case that differs:

- If e_1 is of the form $\mathsf{if}_H\ k\ \mathsf{then}\ e_1^1\ \mathsf{else}\ e_1^0$ and $e_0 \preceq e_1^k$, then e_1 type with a pc $\Upsilon \neq L$. Hence, by Theorem 6, $\langle e^i, \mu^i\rangle_i$ is finite. We conclude with Lemma 8 and 7.

■

Proof: The proof of Theorem 7 is by induction on the number of steps for e to terminate, using Lemma 9. ■

D. Proof of Theorem 8

From Theorem 4, if an expression terminates with two different paths, the memories must be the same. This implies IO non-interference by Theorem 1. We omit the proof for path non-interference since it is similar to the proof of Theorem 9 (with a special case for private functions that use public branches).

E. Proof of Theorem 9

We want to prove equivalence of two executions on memories that differ only on private values. Hence, we first define an equivalence between expressions that share the same public values, but not the privates ones.

Definition 6: Let Γ be an environment, let Υ be a pc, let τ be a type, let e and e' be two expressions such that $\Upsilon, \Gamma \vdash e : \tau$ and $\Upsilon, \Gamma \vdash e' : \tau$. We define the reflexive relation $\sim_{\tau,\Gamma,\Upsilon}$ as:

$$k \sim_{bool^H,\Gamma,\Upsilon} k'$$

plus the natural inductive rules.

We now prove a more generic lemma using this equivalence.

Lemma 10: Let Γ be an environment, let τ be a security type, and let Υ be a pc, let e_0 and e_1 be two expressions such that $e_0 \sim_{\tau,\Gamma,\Upsilon} e_1$, let μ_0 and μ_1 be two pure memories such that, for every $l \in \Gamma$, $\mu_0(l) \sim_{\Gamma(l),\Gamma,\emptyset} \mu_1(l)$. For every memory μ'_0, expression e'_0, and α_0 such that

$$\langle e_0, \mu_0\rangle \xrightarrow{\alpha_0} \langle e_0, \mu'_0\rangle$$

there exist a memory μ'_1, an expression e_1, an environment Γ' with $\Gamma \subseteq \Gamma'$ and α_1 such that

$$\langle e_1, \mu_1\rangle \xrightarrow{\alpha_1} \langle e_1, \mu'_1\rangle$$

$e_0 \sim_{\tau,\Gamma',\Upsilon} e_1$, and for every $l \in \Gamma'$, $\mu'_0(l) \sim_{\Gamma'(l),\Gamma',\emptyset} \mu'_1(l)$.

Proof of Lemma 10: The proof is by structural induction on e_0, with a first pass on pure expressions so that we can apply the lemma on memory reductions.

- If there is a memory reduction in μ_0, we mirror the reduction in μ_1.
- If e_0 is a value, there is no reduction, we conclude.
- If e_0 is of the form $v_0^0 \wedge v_0^1$, then e_1 is of the form $v_0^0 \wedge v_0^1$. Both have only one possible reduction, and we conclude (similarly for $\vee$ and $\neg$.
- If e_0 is of the form $e_0^0\ e_0^1$, then e_1 is of the form $e_1^0\ e_1^1$, with $e_0^0 \sim_{\tau',\Gamma,\Upsilon} e_1^0$ and $e_0^1 \sim_{\tau'',\Gamma,\Upsilon} e_1^1$. If e_0^0 is not a value, we conclude by induction. If the reduction is inside e_0^1 we conclude by induction. Otherwise, e_0^1 is a tree of locations, hence e_1^1 is a tree of locations, the transition can happen on both sides.
- If e_0 is of the form $\mathsf{if}_L\ v_0\ \mathsf{then}\ e_0^0\ \mathsf{else}\ e_0^1$, then e_1 is of the form $\mathsf{if}_L\ v_1\ \mathsf{then}\ e_1^0\ \mathsf{else}\ e_1^1$, with $v_0 \sim_{\tau',\Gamma,\Upsilon} v_1$, $e_0^0 \sim_{\tau'',\Gamma,\Upsilon} e_1^0$, and $e_0^1 \sim_{\tau''',\Gamma,\Upsilon} e_1^1$. There is only one possible reduction, and it works as expected.
- If e_0 is of the form $\mathsf{if}_H\ v_0\ \mathsf{then}\ e_0^0\ \mathsf{else}\ e_0^1$, then e_1 is of the form $\mathsf{if}_H\ v_1\ \mathsf{then}\ e_1^0\ \mathsf{else}\ e_1^1$, with $v_0 \sim_{\tau',\Gamma,\Upsilon} v_1$, $e_0^0 \sim_{\tau'',\Gamma,\Upsilon} e_1^0$, and $e_0^1 \sim_{\tau''',\Gamma,\Upsilon} e_1^1$. If at least one of e_0^0 and e_0^1 is not an expression, there is one or two possible reductions, and they work as expected by induction. If e_0^0 and e_0^1 are both lambda abstraction or booleans, then e_1^0 and e_1^1 also are lambda abstraction; there is one possible reduction, which works correctly. Otherwise, they are both locations and there is no possible reductions.
- If e_0 is of the form $e_0^0 := e_0^1$, then e_1 is of the form $e_1^0 := e_1^1$, with $e_0^0 \sim_{\tau',\Gamma,\Upsilon} e_1^0$ and $e_0^1 \sim_{\tau'',\Gamma,\Upsilon} e_1^1$. If e_0^0 or e_0^1 is not a value or an high if, we conclude by induction. If e_0^0 is an high if, then e_1^0 is too, we verify the reduction. If e_0^1 is an high if, then e_1^1 is too, we verify the reduction. Otherwise, there is one reduction, that works as expected. We conclude.
- We conclude similarly for ! and ref .

■

Proof: The proof of Theorem 9 is direct from Lemma 10. ■