OblivIO: Securing reactive programs by oblivious execution with bounded traffic overheads

Jeppe Fredsgaard Blaabjerg
Aarhus University
jfblaa@cs.au.dk

Aslan Askarov
Aarhus University
aslan@cs.au.dk

***Abstract*—Traffic analysis attacks remain a significant problem for online security. Communication between nodes can be observed by network level attackers as it inherently takes place in the open. Despite online services increasingly using encrypted traffic, the shape of the traffic is not hidden. To prevent traffic analysis, the shape of a system's traffic must be independent of secrets.**

We investigate adapting the data-oblivious approach the reactive setting and present *OblivIO*, a secure language for writing reactive programs driven by network events. Our approach pads with dummy messages to hide which program sends are genuinely executed. We use an information-flow type system to provably enforce timing-sensitive noninterference. The type system is extended with potentials to bound the overhead in traffic introduced by our approach. We address challenges that arise from joining data-oblivious and reactive programming and demonstrate the feasibility of our resulting language by developing an interpreter that implements security critical operations as constant-time algorithms.

I. Introduction

Online communication between network nodes inherently takes place in the open. Even when using encrypted traffic, network level attackers can still observe the traffic shape including timing, bandwidth, and destination. Traffic analysis attacks are possible when secrets affect the traffic shape of a system and enable attackers to infer often highly detailed, sensitive information about user interactions across a wide range of online services [1].

Strategies for mitigating traffic analysis have predominantly been developed at the system-level [2] where solutions can be applied to existing applications and services in a black-box fashion. However, this approach incurs significant – if not practically infeasible – overheads in latency and bandwidth in order to achieve a high degree of security [3]. System-level approaches are program-agnostic by nature and hence performs some traffic padding even when no possible control flow would produce genuine traffic.

Traffic analysis is made possible by observable attributes of shape of traffic, such as time. Timing leaks have been extensively studied in the programming languages community. Language-based approaches are particularly appealing as language-level information can make the enforcement precise. Compared with general approaches for mitigating timing channels [4, 5, 6, 7], our focus on traffic analysis enables us to exploit dummy traffic as a means to reduce overheads introduced by the enforcement.

```
1 TRANSFER(from: int,amount: int,to: int) {
2   if amount <= balance[from]
3   then {
4     balance[from] -= amount;
5     balance[to]   += amount;
6   }
7   else send(ERROR,(amount,balance[from]));
8 }
```

Listing 1. Traffic leak

Timing channels are also a significant security concern for the cryptographic community. Almost all modern cryptography deals with timing side-channels by *constant-time* programming [8]. Constant-time programs may neither branch nor access memory depending on secrets. This ensures that programs do not leak via their execution time. However, these restrictions make adhering to constant-time programming hard and requires developers to deviate from conventional programming practices [9]. Data-oblivious languages [10, 11] ease the restrictions imposed by constant-time programming by introducing oblivious conditionals and oblivious memory accesses. Oblivious conditionals execute both branches, but negate the unwanted side-effects of the non-chosen branch. The data-oblivious approach does not permit loops with secret guards as loop termination would cause a leak. Instead, the programmer can provide a public upper bound on the number of loop iterations and obliviously branch on secrets inside the loop.

The reactive programming model is a natural fit for online services and IoT devices. In this model, applications are written as a collection of event handlers that are triggered in response to observing associated events. This allows for scalable, reliable software, as it assumes no control over, or knowledge of, the number or timing of events [12]. However, as shown by Chen et al. [1], many applications written in this model leak sensitive information. Listing 1 shows a simple handler for transferring money from one account to another, for example as part of an online transaction. If the amount to transfer exceeds the balance of the source account, no money is transferred and an error message is sent. This message will be observable if sent over standard internet protocols.

In this paper we consider the problem of mitigating traffic analysis in a reactive setting. We consider traffic leaks through the observable timing, size, and destination of messages. Our strategy for preventing traffic leaks is to adapt the data-oblivious approach to the reactive setting: execute both

branches of sensitive conditionals, pad the traffic with dummy messages at send commands in non-chosen branches, and obliviously execute handlers upon receiving dummy messages. We implement this strategy in the design of *OblivIO*, a secure language for reactive programs, driven by network events. Programs written in the language are secure by construction, satisfying progress-sensitive, timing-sensitive noninterference, while incurring a bounded overhead in the number of dummy messages. The main contributions of this paper are:

- We adapt the data-oblivious approach to the reactive setting and develop *OblivIO*, a statically typed language for writing reactive programs.
- We provide a security-labelled type system and prove that well-typed programs in *OblivIO* are secure against network level attackers, satisfying progress-sensitive, timing-sensitive noninterference.
- We bound the traffic overhead introduced by our approach by typing commands and network channels with potentials and show that the overhead is bounded by the potentials.
- We demonstrate the practicality of our approach by developing a real-time interpreter that implements oblivious branching and constant-time algorithms for security critical operations.

Our model builds upon the work of Bohannon et al. [13] that gives the foundational model for noninterference for reactive programs. They develop a type system that enforces timing- and termination-insensitive noninterference. Their model does not consider leaks to network level attackers.

Like Bohannon et al. [13], we rely on static enforcement of noninterference. Static enforcement is well-suited for our problem as both the code and the types of reactions in the program are well-known and as we enforce progress-sensitive security. Here, we benefit from the precision of static analysis with regards to termination behaviour, which is difficult to achieve in purely dynamic setting. Many dynamic approaches use hybrid techniques where they inspect non-chosen branches of conditionals to recover this precision [14, 15]. Static enforcement would not be suitable if code or the types of reactions were not statically known.

The rest of this paper is structured as follows. In Section II we discuss how we adapt the data-oblivious approach to the reactive setting and provide the semantics of *OblivIO*. The type system is presented in Section III. Section IV presents the formal security condition and noninterference theorem, while Section V presents a bound on the traffic overhead introduced by our approach. In Section VI we demonstrate the language by example. We discuss how we have implemented the language semantics in our interpreter in Section VII. We discuss our approach and compare it with existing approaches in Section VIII and provide related work in Section IX before we conclude in Section X.

II. Language

In this section we provide brief background on reactive programs and data-obliviousness and discuss the terminology we will use in this paper. We then give our threat model and provide the semantics of *OblivIO*.

A. Background and terminology

We briefly outline reactive programs, constant-time programming, and data-obliviousness, and the distinction between genuine and dummy traffic.

Reactive programs: Reactive programs consist of handlers that are triggered by associated events, imposing no structure on the order or timing of events. This allows for reliable programs that scale easily by introducing new events and handlers.

Reactive programming is a popular model for online services, where client interaction is intermittent and irregular, due to its inherent flexibility, and languages such as JavaScript follow this model. Online services present radically different security challenges from applications running on a single machine or a closed network as communication channels use public, potentially compromised infrastructure.

Constant-time programming: Constant-time programming is at the heart of almost all modern cryptography [8, 16, 17]. Under this paradigm, programs may not branch on secrets, access secret memory locations, or use secrets in variable-time operations such as division. This ensures that programs are secure, but makes them difficult for developers to write [9].

Data-oblivious programming: Data-oblivious programming relaxes some of the restrictions imposed by constant-time programming. The principal idea of data-oblivious languages is oblivious conditionals [10, 11]. Instead of a branch in the control-flow, oblivious conditionals execute both branches and negate the unwanted side-effects in the non-chosen branch. This prevents branching-related timing differences and ensures that secrets do not affect whether a command is executed or whether an expression is evaluated.

Dummy messages: A common strategy for mitigating traffic analysis is to introduce dummy messages. Dummy messages are additional, fake messages introduced by the enforcement mechanism [3, 18, 19, 20]. Informally, dummy messages should only serve to hide which traffic is genuine and should not not alter the semantics of a system. This does not always hold in practice, e.g., as dummy messages may introduce overhead in latency or execution time that affects time reads.

B. A data-oblivious approach to reactive programming

Extending the data-oblivious approach to programs with observable IO is not straightforward as observable side-effects cannot safely be suppressed. If an assign statement occurs in a sensitive conditional, the assignment must be suitably padded in order to hide the genuine branch. Otherwise, the value would be unsafe to send at any later point, as a network level attacker could observe the size of the value and thus infer the genuine branch. Suppressing messages in non-chosen branches is also not an option as the absence of traffic would be observed. Instead, any message sent in a non-chosen branch must be replaced a convincing dummy message. Additionally, such dummy message must be reacted to and handled in a convincing manner, including possibly generating new traffic.

Combining reactive and data-oblivious programming therefore introduces the need for bounding the number of dummy messages generated by a system. To demonstrate the problem, we consider the interaction between the two small programs below, where `oblif` is a primitive for oblivious branching. For every message received, two new messages are sent. If this program were allowed, the amount of dummy traffic would grow exponentially over time, dominating real computation and crippling the utility of the system.

```
1 PING (x: int) {
2   oblif x
3   then send(PONG,1);
4   else send(PONG,0);
5 }
```

```
1 PONG (x: int) {
2   oblif x
3   then send(PING,1);
4   else send(PING,0);
5 }
```

To address this issue, we take inspiration from static resource analysis, where types are annotated with potentials [21, 22, 23]. We discuss how we use potentials in Section III and show a bound on the overhead in traffic in Section V.

C. Threat model

We consider the security of a network node running a reactive program that processes incoming network messages sequentially using a single event loop. Every handler in the program defines a channel endpoint, and other nodes can send messages on the associated channel to trigger execution of the handler. We assume that all nodes run *OblivIO* programs. We consider a lattice $\mathcal{L}$ of security levels ℓ and assign levels to each channel. The lattice has distinguished bottom element $\bot$, corresponding to the network level.

We consider an active adversary who can be one of the other network nodes. The adversary knows the program being run and knows its initial secrets up to some level ℓ_{adv}. We assume that the adversary can observe the presence, size, time, and channel of all network messages, can drop messages, and perform replay attacks. We assume that the contents of network messages is hidden from unprivileged parties by encryption, and that the adversary can decrypt and read the contents of messages sent on channels up to level ℓ_{adv}.

D. Language

OblivIO is a simple, statically typed language for data-oblivious, reactive programs. Programs written in *OblivIO* consist of a number of handlers that are triggered by associated events. We let events model network messages. Each handler in a program defines a channel endpoint, such that messages sent on the channel get processed by the handler. Events are processed sequentially and during execution, *OblivIO* programs change between consumer states, C, that wait for the next network message, and producer states, P, that execute the appropriate handler for the current network message. We model incoming network messages using strategies, functions that map network traces to messages.

We let n range over integer literals and s range over string literals. We let x range over variables. Values in *OblivIO* are written $(\!|v|\!)_z$ and consist of a base value v and a size z. Base values v are either an integer or a string and sizes z are non-negative integers. We assume an attacker that can observe the size of any network message sent. Therefore, we cannot easily protect secrets of arbitrary size. Instead, we settle for protecting secrets up to a publicly known upper bound. This upper bound can be any size greater than or equal to the actual size of the secret. That is, we assume that value $(\!|v|\!)_z$ can be padded to $(\!|v|\!)_{z'}$ for any $z' \geq z$. We assume a function *size* for computing the size of base values v and maintain well-formed values such that for any $(\!|v|\!)_z$ we have $size(v) \leq z$. We provide the formal definition in Section III.

At runtime, the system maintains a persistent, global store μ mapping variables to size-annotated values. Handlers define a variable x for binding the value of a received message as a read-only value in local memory m. Local memory m is cleared when the execution of the handler finishes.

We assume two distinct types of channels: local and network. Local channels model non-network channels, where the presence of messages is not observable to the attacker, e.g., keyboard input or sensor readings. We maintain a queue for each local channel to prevent the presence of messages on one channel from tainting reads on another. We let π denote a local environment: a mapping from local channels to a stream of value options. We let $\bullet$ denote that no value is available. We choose this representation as it makes it easier to bound the overhead in traffic (Section V). Network channels model channels where communication can be observed. Thus, the presence of messages on all network channels is public. This allows network messages to be combined into a single queue. We model incoming network traffic using network strategies ω, functions from network traces to messages of form $ch(t, b, (\!|v|\!)_z)$. We choose this model over the equally expressive model of streams ([24]) as it does not pre-compute future messages, thereby more intuitively capturing the interactive nature of network communication and allowing us to more faithfully state our overhead theorem (Theorem 2).

The distinction between local and network channels is motivated by the observation that messages with secret presence must be handled specially. Observable side-effects such as sending message or introducing latency would leak the presence of such message. We therefore minimise the interface with messages with secret presence by not considering them events that handlers react to. Instead, we allow programs to read from local channels using input statements.

We now explain the formal semantics and non-standard features of the language. We assume a security lattice $\mathcal{L}$ of security levels ℓ with bottom element $\bot$, lattice ordering $\sqsubseteq$, and least upper bound operation $\sqcup$. Level $\bot$ corresponds to the network level. We also refer to this level as *public*. We assume that all nodes run *OblivIO* programs. We assume that message contents can be sufficiently hidden, e.g., using encryption, but that their presence, size, and destination are publicly observable.

Figure 1 presents the syntax of *OblivIO*. A program p consists of a number of handlers $ch(x)\{c\}$, where ch is the network channel associated with the handler and variable x binds the incoming message in local memory. Commands c

are largely standard, though with novel commands to facilitate oblivious execution. Commands `stop` and `pop` are only used internally and therefore not part of the syntax of the language.

$$
\begin{aligned}
p &::= \cdot \mid ch(x)\{c\}; p \\
c &::= \texttt{skip} \mid c; c \mid x = e \mid x \texttt{ ?= } e \mid x \texttt{ ?= input}(ch, e) \\
&\quad \mid \texttt{send}(ch, e) \mid \texttt{if } e \texttt{ then } c \texttt{ else } c \\
&\quad \mid \texttt{while } e \texttt{ do } c \mid \texttt{oblif } e \texttt{ then } c \texttt{ else } c \\
e &::= n \mid s \mid x \mid e \oplus e \\
\oplus &::= + \mid - \mid \star \mid = \mid \texttt{!=} \mid < \mid \texttt{<=} \mid > \mid \texttt{>=} \mid \texttt{\&\&} \mid \texttt{||} \mid \hat{}
\end{aligned}
$$

Figure 1. Syntax of the language

We use a big-step semantics for evaluating expressions (Figure 2). Expression e are evaluated using local memory m and global store μ. For simplicity, memory m is read-only in the presented model and binds just the current message value in the handler variable. We let local memory shadow the global store.

$$
\frac{size(n) = z}{\langle n, m, \mu\rangle \Downarrow (\!|n|\!)_z} \qquad
\frac{size(s) = z}{\langle s, m, \mu\rangle \Downarrow (\!|s|\!)_z} \qquad
\frac{x \in dom(m)}{\langle x, m, \mu\rangle \Downarrow m(x)}
$$

$$
\frac{x \notin dom(m)}{\langle x, m, \mu\rangle \Downarrow \mu(x)} \qquad
\frac{\langle e_1, m, \mu\rangle \Downarrow (\!|v_1|\!)_{z_1} \quad \langle e_2, m, \mu\rangle \Downarrow (\!|v_2|\!)_{z_2} \quad v_1 \oplus v_2 = v_3 \quad z_1 \oplus_{size} z_2 = z_3}{\langle e_1 \oplus e_2, m, \mu\rangle \Downarrow (\!|v_3|\!)_{z_3}}
$$

Figure 2. Semantics of evaluating expressions

We assume that binary operations $\oplus$ are total and have an associate operation $\oplus_{size}$ for computing the size of the result from the sizes of the operands. We further assume that binary operations preserve well-formed values, that is, if $size(v_1) \leq z_1$, $size(v_2) \leq z_2$, and $v_1 \oplus v_2 = v_3$, then $size(v_3) \leq z_1 \oplus_{size} z_2$. We discuss how $\oplus_{size}$ can be implemented in Section VII.

E. Command semantics

Commands are evaluated using a small-step operational presented in Figure 3. Configuration $\langle \overline{b}, c, m, \mu, \pi, h\rangle$ consists of a command c, memory m, store μ, local environment π, and history h. We let history h record the commands executed in a run and the variables used. This allows us to model that different instructions take different amounts of time as well as cache- and branch-related timing differences [5, 25]. We assume that execution time of instructions is affected by the size of values used, but not by their specific value. This assumption does not come for free, but requires careful handling of sensitive language primitives. Extending our language with pointers, arrays, or variable-time operations would require careful consideration. One straightforward strategy for sensitively indexing arrays is by linear scan over all array elements and using bit-masks to select only the element at the desired index.

To obtain high-resolution timestamps t, we assume a strictly increasing function *time* from histories to numeric values representing real time. That is, for all histories h and history events ev we have $time(h) < time(h :: ev)$. Histories and high-resolution timestamps may appear too strong a formalism considering that we execute sensitive conditionals obliviously and therefore have that runs that agree on initial public state will agree on history and time. This level of accuracy is motivated by our attacker model and allows us to show the strong security guarantee enforced by our approach.

We let $\overline{b}$ denote a stack of execution mode bits $b \in \{1, 0\}$. For consistency with existing terminology [11] we say that execution takes place in either *real* or *phantom* mode. We let 1 denote real mode and 0 denote phantom mode. The execution mode is first set when triggering a handler. Genuine messages are handled in real mode while dummy messages are handled in phantom mode. The execution mode changes from real to phantom mode for the non-chosen branch when obliviously branching by `oblif`.

Transitions $\langle \overline{b}, c, m, \mu, \pi, h\rangle \xrightarrow{\alpha} \langle \overline{b}', c', m', \mu', \pi', h'\rangle$ denote a step from one configuration to another emitting output event α. Output events α are possibly empty, denoted ϵ, and are given by the following grammar:

$$\alpha ::= \epsilon \mid \overrightarrow{ch}(t, b, (\!|v|\!)_z)$$

Non-empty events correspond to network traffic and contain channel ch, denoting which channel the message is sent on; high-resolution timestamp t, denoting when the message is sent; bit b, denoting the execution mode the message was sent under; and value $(\!|v|\!)_z$. Timestamps t are attacker observable giving us a strong attacker model (Section IV). Intuitively, programs are noninterferent if related runs agree on the timestamps of all messages, which we can only ensure if their computational histories are the same. Bit b indicate whether a message is genuine or dummy. Messages produced in real mode $b = 1$ are genuine, while messages produced in phantom mode $b = 0$ are dummy. This enables the recipient of the message to execute the handler in the appropriate mode. We now explain the formal semantics and non-standard features.

If: Standard conditionals are done by `if` e `then` c_1 `else` c_2. The rule is largely standard, but to accurately model branch-related timing effects we append history h with event $\mathsf{br}(e, z, i)$, where i is the chosen branch.

Oblif and Pop: We introduce command `oblif` e `then` c_1 `else` c_2 for oblivious branching. When obliviously branching, two bits are pushed onto bit-stack $\overline{b}$, one for each branch. The chosen branch continues with the current execution mode b, while the non-chosen branch is executed in phantom mode 0. After executing each branch, command `pop` removes the top element of the bit-stack.

Assign and oblivious assign: Standard assignment is done by command $x = e$. Standard assignment is intended for use only in real mode, and we do not give semantics to the command for phantom mode. Our type system enables us to ensure that the command is never reached in phantom mode (Section III). We introduce a command x `?=` e for oblivious assignment. The command conditionally updates the base value of x depending on the execution mode and unconditionally pads the size of the value in the store to mask whether the base value was changed.

Skip

$$\frac{}{\langle \overline{b}, \texttt{skip}, m, \mu, \pi, h\rangle \longrightarrow \langle \overline{b}, \texttt{stop}, m, \mu, \pi, h :: \mathsf{skp}\rangle}$$

Seq1

$$\frac{\langle \overline{b}, c_1, m, \mu, \pi, h\rangle \xrightarrow{\alpha} \langle \overline{b}', c_1', m', \mu', \pi', h'\rangle \qquad c_1' \neq \texttt{stop}}{\langle \overline{b}, c_1; c_2, m, \mu, \pi, h\rangle \xrightarrow{\alpha} \langle \overline{b}', c_1'; c_2, m', \mu', \pi', h'\rangle}$$

Seq2

$$\frac{\langle \overline{b}, c_1, m, \mu, \pi, h\rangle \xrightarrow{\alpha} \langle \overline{b}', \texttt{stop}, m', \mu', \pi', h'\rangle}{\langle \overline{b}, c_1; c_2, m, \mu, \pi, h\rangle \xrightarrow{\alpha} \langle \overline{b}', c_2, m', \mu', \pi', h'\rangle}$$

Assign

$$\frac{\langle e, m, \mu\rangle \Downarrow (\!|v|\!)_z \qquad \mu' = \mu[x \mapsto (\!|v|\!)_z]}{\langle 1 :: \overline{b}, x = e, m, \mu, \pi, h\rangle \longrightarrow \langle 1 :: \overline{b}, \texttt{stop}, m, \mu', \pi, h :: \mathsf{asn}(x, e, z)\rangle}$$

OblivAssign

$$\frac{\mu(x) = (\!|v_0|\!)_{z_0} \qquad \langle e, m, \mu\rangle \Downarrow (\!|v_1|\!)_{z_1} \qquad z = max(z_0, z_1) \qquad i = \begin{cases} 1 & if\ b = 1 \\ 0 & if\ b = 0 \end{cases}}{\langle b :: \overline{b}, x\ \texttt{?=}\ e, m, \mu, \pi, h\rangle \longrightarrow \langle b :: \overline{b}, \texttt{stop}, m, \mu[x \mapsto (\!|v_i|\!)_z], \pi, h :: \mathsf{casn}(x, e, z)\rangle}$$

LocalInput

$$\frac{\mu(x) = (\!|v_x|\!)_{z_x} \qquad \langle e, m, \mu\rangle \Downarrow (\!|n_e|\!)_{z_e} \qquad z' = max(z_x, n_e) \qquad v', \pi' = \begin{cases} v, \pi[ch \mapsto tl] & \text{if } b = 1 \text{ and } \pi(ch) = (\!|v|\!)_z :: tl \text{ and } z \leq n_e \\ v_x, \pi[ch \mapsto tl] & \text{if } b = 1 \text{ and } \pi(ch) = \bullet :: tl \\ v_x, \pi & \text{otherwise} \end{cases}}{\langle b :: \overline{b}, x\ \texttt{?=}\ \texttt{input}(ch, e), m, \mu, \pi, h\rangle \longrightarrow \langle b :: \overline{b}, \texttt{stop}, m, \mu[x \mapsto (\!|v'|\!)_{z'}], \pi', h :: \mathsf{in}(x, ch, e, z_x)\rangle}$$

Send

$$\frac{\langle e, m, \mu\rangle \Downarrow (\!|v|\!)_z \qquad h' = h :: \mathsf{out}(ch, e, z) \qquad t = time(h')}{\langle b :: \overline{b}, \texttt{send}(ch, e), m, \mu, \pi, h\rangle \xrightarrow{\overrightarrow{ch}(t, b, (\!|v|\!)_z)} \langle b :: \overline{b}, \texttt{stop}, m, \mu, \pi, h'\rangle}$$

If

$$\frac{\langle e, m, \mu\rangle \Downarrow (\!|v|\!)_z \qquad v \neq 0 \implies i = 1 \qquad v = 0 \implies i = 2}{\langle \overline{b}, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m, \mu, \pi, h\rangle \longrightarrow \langle \overline{b}, c_i, m, \mu, \pi, h :: \mathsf{br}(e, z, i)\rangle}$$

While

$$\frac{c' = \texttt{if}\ e\ \texttt{then}\ c\texttt{;}\ \texttt{while}\ e\ \texttt{do}\ c\ \texttt{else skip}}{\langle 1 :: \overline{b}, \texttt{while}\ e\ \texttt{do}\ c, m, \mu, \pi, h\rangle \longrightarrow \langle 1 :: \overline{b}, c', m, \mu, \pi, h :: \mathsf{whl}\rangle}$$

Pop

$$\frac{}{\langle b :: \overline{b}, \texttt{pop}, m, \mu, \pi, h\rangle \longrightarrow \langle \overline{b}, \texttt{stop}, m, \mu, \pi, h :: \mathsf{pop}\rangle}$$

OblivIf

$$\frac{\overline{b} = b :: \_ \qquad \langle e, m, \mu\rangle \Downarrow (\!|v|\!)_z \qquad v \neq 0 \implies b_1 = b \wedge b_2 = 0 \qquad v = 0 \implies b_1 = 0 \wedge b_2 = b \qquad h' = h :: \mathsf{obr}(e, z)}{\langle \overline{b}, \texttt{oblif}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m, \mu, \pi, h\rangle \longrightarrow \langle b_1 :: b_2 :: \overline{b}, c_1; \texttt{pop}; c_2; \texttt{pop}, m, \mu, \pi, h'\rangle}$$

Figure 3. Operational semantics of commands

Input: Programs sample local channels using non-blocking input command x `?=` $\texttt{input}(ch, e)$. Receives on channels with secret presence are restricted to non-blocking semantics as blocking would cause a leak [26]. The value of variable x is only changed if executed in real mode and a message is available with size less than the evaluation of expression e. The size of $\mu(x)$ is unconditionally padded, similar to oblivious assignment. The head of the message stream associated with the local channel is either $(\!|v|\!)_z$, if a value is available, or $\bullet$, if no value is available.

Send: Command $\texttt{send}(ch, e)$ evaluates expression e to obtain value $(\!|v|\!)_z$. The step emits event $ch(t, b, (\!|v|\!)_z)$ capturing the network message; the value, when and to whom it was sent, and under what mode.

While: Since the base values in store μ and memory m are not modified while under phantom mode, executing loops in phantom mode would lead to non-termination as the value of guard expression e could not be modified. For this reason, we restrict while-loops to real-mode execution.

F. Program semantics

A reactive program is at any point in one of two states. Consumer states $(p, \mu, \pi, \omega, h, \tau)$ consist of program p, global store μ, local environment π, network strategy ω, history h, and trace τ. Consumer states, as the name implies, consumes network events and triggers the appropriate handlers. Producer states $(p, \overline{b}, c, m, \mu, \pi, \omega, h, \tau)^{ch}$ consist of program p, bit-stack $\overline{b}$, command c, local memory m, global store μ, local environment π, network strategy ω, history h, trace τ, and are annotated with channel ch associated with the currently executing handler. We let C range over consumer states, P range over producer states, and Q range over both consumer and producer states.

We extend events α with incoming messages $\overleftarrow{ch}(t, b, (\!|v|\!)_z)$ and observed messages $\widetilde{ch}(t, b, (\!|v|\!)_z)$:

$$\alpha ::= \ldots \mid \overleftarrow{ch}(t, b, (\!|v|\!)_z) \mid \widetilde{ch}(t, b, (\!|v|\!)_z)$$

Observed messages $\widetilde{ch}(t, b, (\!|v|\!)_z)$ correspond to the local node observing the network traffic between two remote nodes. Their inclusion enables us to bound the traffic overhead for the entire network (Section V). We let $\rightsquigarrow$ range over $\{\leftarrow, \rightarrow, \sim\}$.

Network traces τ are used as input for strategies ω to obtain the next network message and are given by the following grammar:

$$\tau ::= \epsilon \mid \tau \cdot \alpha$$

By convention, we only modify the trace when appending non-empty events, that is, for any τ we have $\tau \cdot \epsilon = \tau$.

We now give semantics to the states and state transitions of a reactive program (Figure 4).

CC

$$\frac{\omega(\tau) = ch(t, b, (\!|v|\!)_z) \qquad (p)(ch) \not\Downarrow}{(p, \mu, \pi, \omega, h, \tau) \longrightarrow (p, \mu, \pi, \omega, h, \tau \cdot \widetilde{ch}(t, b, (\!|v|\!)_z))}$$

CP

$$\frac{\omega(\tau) = ch(t, b, (\!|v|\!)_z) \quad (p)(ch) \Downarrow c, x \qquad h' = h :: \mathsf{hl}(ch, t, z) \qquad \tau' = \tau \cdot \overleftarrow{ch}(t, b, (\!|v|\!)_z)}{(p, \mu, \pi, \omega, h, \tau) \longrightarrow (p, [b], c, [x \mapsto (\!|v|\!)_z], \mu, \pi, \omega, h', \tau')^{ch}}$$

PP

$$\frac{\langle \overline{b}, c, m, \mu, \pi, h\rangle \xrightarrow{\alpha} \langle \overline{b}', c', m', \mu', \pi', h'\rangle}{(p, \overline{b}, c, m, \mu, \pi, \omega, h, \tau)^{ch} \longrightarrow (p, \overline{b}', c', m', \mu', \pi', \omega, h', \tau \cdot \alpha)^{ch}}$$

PC

$$\frac{}{(p, \overline{b}, \texttt{stop}, m, \mu, \pi, \omega, h, \tau)^{ch} \longrightarrow (p, \mu, \pi, \omega, h :: \mathsf{ret}, \tau)}$$

Figure 4. Operational semantics of system

Consumer state C transitions depending on whether program p defines a handler associated with that channel (Figure 5).

$$\frac{}{(ch(x)\{c\}; p)(ch) \Downarrow c, x} \qquad \frac{ch \neq ch' \qquad (p')(ch') \Downarrow c', x'}{(ch(x)\{c\}; p')(ch') \Downarrow c', x'}$$

Figure 5. Handler selection

If p defines no associated handler, written $(p)(ch) \not\Downarrow$, execution continues in consumer state C', where the trace has been annotated with an observed message $\widetilde{ch}(t, b, (\!|v|\!)_z)$. If p defines an associated handler, written $(p)(ch) \Downarrow c, x$, handler selection evaluates to command c and variable x. Local memory m is constructed by assigning message value $(\!|v|\!)_z$ to variable x and execution proceeds in producer state P, executing command c. The producer state executes in the mode b of the received message by using singleton bit-stack $[b]$. A producer state P steps by the operational semantics of commands (Figure 3), appending emitted events to the trace. Execution continues in producer state until reaching command `stop`, when it transitions back into consumer state.

III. Enforcement

In this section we provide and discuss the type system for *OblivIO*. We type global stores μ using typing environment Γ. We type local channels using typing environment Π and type network channels using typing environment Λ. Typing environments Γ, Π, and Λ are static and do not change during execution. Local memory m is typed using typing environment Δ which is computed per handler. We give variables in store and memory, and local channels, a type of the form $\sigma@\ell$, where $\sigma \in \{int, string\}$ and $\ell \in \mathcal{L}$. We let q, r range over non-negative integer potentials and give network channels type of the form $\sigma@\ell_{mode}; \ell_{val}; q$, where ℓ_{mode} is the security level of the message mode, and ℓ_{val} is the security level of the message value. Potentials q, r are inspired by static resource analysis [21, 22, 23]. Resource analysis commonly uses potentials to infer the resource bounds of a program. We use potentials q to bound the number of dummy messages that may be generated by the handler of the channel.

A. Typing of expressions

We now present the type system for *OblivIO*. Expressions are typed $\Gamma; \Delta \vdash e : \sigma@\ell$ (Figure 6). We type local memory using environment Δ and type global store using environment Γ. We let bindings in Δ shadow bindings in Γ. The rules are otherwise standard.

$$\frac{}{\Gamma; \Delta \vdash n : int@\ell} \qquad \frac{}{\Gamma; \Delta \vdash s : string@\ell} \qquad \frac{x \in dom(\Delta)}{\Gamma; \Delta \vdash x : \Delta(x)}$$

$$\frac{x \notin dom(\Delta)}{\Gamma; \Delta \vdash x : \Gamma(x)} \qquad \frac{\oplus : \sigma_1 \times \sigma_2 \to \sigma_3 \quad \Gamma; \Delta \vdash e_1 : \sigma_1@\ell_1 \qquad \Gamma; \Delta \vdash e_2 : \sigma_2@\ell_2}{\Gamma; \Delta \vdash e_1 \oplus e_2 : \sigma_3@\ell_1 \sqcup \ell_2}$$

Figure 6. Typing rules for expressions

B. Typing of commands

Commands are typed $\Gamma, \Pi, \Lambda; \Delta; pc \vdash^q c$. Potential q bounds the number of dummy messages that can be produced while executing command c, as well as any additional dummy messages that are transitively produced across the network when handling dummy messages produced by executing c. ASThis allows us to reason locally about the network-wide overhead in traffic. The typing rules are provided in Figure 7.

Our key insight is to restrict program commands with observable side-effects and provide safe, oblivious counterparts where possible. In standard models, label pc tracks the security level of the control flow. That is not quite so in our model. The semantics of obliviously branching means that the control flow is the same for all executions that agree on the initial public state. However, the execution mode may differ. As such, pc tracks the security level of the execution mode in our model. We enforce that phantom mode only occurs for commands that type with non-public pc.

Sequential composition: Sequential composition $c_1; c_2$ types with potential $q_1 + q_2$ if c_i types with potential q_i, for $i = 1, 2$. As $q_1, q_2 \geq 0$ this prevents double spending of potential.

Assign and oblivious assign: We allow standard, unconditional assignment only under public pc as we only give semantics to the command when executing in real mode. We allow oblivious assignment under any pc. Rule T-OblivAssign is similar to standard assignment rules in other models and is made safe by the padding semantics of OblivAssign.

$$\text{T-Skip}\quad \frac{}{\Gamma, \Pi, \Lambda; \Delta; pc \vdash^{q} \texttt{skip}}$$

$$\text{T-Assign}\quad \frac{x \notin dom(\Delta) \qquad \Gamma(x) = \sigma@\ell_x \qquad \Gamma; \Delta \vdash e : \sigma@\ell_e \qquad \ell_e \sqsubseteq \ell_x}{\Gamma, \Pi, \Lambda; \Delta; \bot \vdash^{q} x = e}$$

$$\text{T-Seq}\quad \frac{\Gamma, \Pi, \Lambda; \Delta; pc \vdash^{q_1} c_1 \qquad \Gamma, \Pi, \Lambda; \Delta; pc \vdash^{q_2} c_2}{\Gamma, \Pi, \Lambda; \Delta; pc \vdash^{q_1+q_2} c_1; c_2}$$

$$\text{T-OblivAssign}\quad \frac{x \notin dom(\Delta) \qquad \Gamma(x) : \sigma@\ell_x \qquad \Gamma; \Delta \vdash e : \sigma@\ell_e \qquad \ell_e \sqcup pc \sqsubseteq \ell_x}{\Gamma, \Pi, \Lambda; \Delta; pc \vdash^{q} x \ ?= e}$$

$$\text{T-LocalInput}\quad \frac{x \notin dom(\Delta) \qquad \Gamma(x) : \sigma@\ell_x \qquad \Pi(ch) = \sigma@\ell_{ch} \qquad \Gamma; \Delta \vdash e : int@\ell_e \qquad \ell_e \sqcup pc \sqsubseteq \ell_{ch} \sqsubseteq \ell_x}{\Gamma, \Pi, \Lambda; \Delta; pc \vdash^{q} x \ ?= \texttt{input}(ch, e)}$$

$$\text{T-Send}\quad \frac{\Gamma; \Delta \vdash e : \sigma@\ell_e \qquad \Lambda(ch) = \sigma@\ell_{mode}; \ell_{val}; r \qquad pc \sqsubseteq \ell_{mode} \qquad \ell_e \sqsubseteq \ell_{val} \qquad q' = \begin{cases} 0 & \text{if } pc = \bot \\ 1 + r & \text{otherwise} \end{cases}}{\Gamma, \Pi, \Lambda; \Delta; pc \vdash^{q+q'} \texttt{send}(ch, e)}$$

$$\text{T-If}\quad \frac{\Gamma; \Delta \vdash e : int@\bot \qquad \Gamma, \Pi, \Lambda; \Delta; pc \vdash^{q} c_1 \qquad \Gamma, \Pi, \Lambda; \Delta; pc \vdash^{q} c_2}{\Gamma, \Pi, \Lambda; \Delta; pc \vdash^{q} \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2}$$

$$\text{T-While}\quad \frac{\Gamma; \Delta \vdash e : int@\bot \qquad \Gamma, \Pi, \Lambda; \Delta; \bot \vdash^{0} c}{\Gamma, \Pi, \Lambda; \Delta; \bot \vdash^{q} \texttt{while } e \texttt{ do } c}$$

$$\text{T-OblivIf}\quad \frac{\Gamma; \Delta \vdash e : int@\ell \qquad \ell \neq \bot \qquad \Gamma, \Pi, \Lambda; \Delta; pc \sqcup \ell \vdash^{q_1} c_1 \qquad \Gamma, \Pi, \Lambda; \Delta; pc \sqcup \ell \vdash^{q_2} c_2}{\Gamma, \Pi, \Lambda; \Delta; pc \vdash^{q_1+q_2} \texttt{oblif } e \texttt{ then } c_1 \texttt{ else } c_2}$$

Figure 7. Typing of commands

If and oblif: We restrict non-oblivious conditionals to public guards only and type $\texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2$ with potential q such that q types both c_1 and c_2. For branching on secrets, we use oblivious branching. The potential annotations in typing rule T-OblivIf resemble those of T-Seq as both branches are executed and and both branches may produce traffic. We restrict oblivious branching to non-public guards to ensure that phantom computation only takes place under non-public pc.

While: Like other data-oblivious languages [10, 11], we restrict while-loops to public guards. Unlike conditionals, there is no oblivious counterpart to while-loops. Intuitively, if the guard of a while-loop were secret, ending the loop at any point would cause a leak. A common technique in oblivious programming is to use a public upper-bound as the guard for the loop and obliviously branch on the secret inside the loop [10, 11]. *OblivIO* allows programmers to adopt this approach. Rule T-While further restricts that $\texttt{while } e \texttt{ do } c$ must be typed with $pc = \bot$ and that c must be typed with potential $q = 0$. The restriction on pc helps us ensure that computation in phantom mode terminates. We do not update base values in memory or store in phantom mode, thus guard e would evaluate to the same value in every iteration of the loop if executed under phantom mode. The restriction on potential is motivated by bounding the number of dummy messages through the type system and prevents double spending of potential.

Send: For sends, typing rule T-Send enforces that the program counter label pc flows to context label ℓ_{mode} of the receiving channel, and that expression label ℓ_e flows to the value label ℓ_{val}. Our type system and semantic rules ensure that only commands that can be typed with non-public $pc \neq \bot$ may be executed in phantom mode. Labelling channels with $\ell_{mode} = \bot$ therefore ensures that only genuine messages may be sent on the channel and allows the handler to perform non-oblivious assignments and while-loops even when $\ell_{val} \neq \bot$.

T-Send requires that $1 + r$ potential is available if $pc \neq \bot$, where r is the potential available in the recipient handler of the message. This accounts for the message possibly being dummy as well as dummy messages produced transitively across the network in response. Because dummy messages require strictly greater potential than what is available in the recipient handler (as they can only be sent under non-public pc), trace potential is strictly decreasing when sending dummy messages. This prevents handlers from sending dummy messages in an infinite loop. Note that loops are possible under $pc = \bot$ where we are assured that messages are genuine. We show that potentials bound the number of dummy messages in Section V.

C. Typing of programs and systems

Program p is well-typed if the body of each handler is well-typed with respect to their respective channel (Figure 8).

$$\frac{}{\Gamma, \Pi, \Lambda \vdash \cdot} \qquad \frac{\Lambda(ch) = \sigma@\ell_{mode}; \ell_{val}; q \qquad \Gamma, \Pi, \Lambda; [x \mapsto \sigma@\ell_{val}]; \ell_{mode} \vdash^{q} c \qquad \Gamma, \Pi, \Lambda \vdash p}{\Gamma, \Pi, \Lambda \vdash ch(x)\{c\}; p}$$

Figure 8. Typing of programs

We formally define well-formed values (Definition 1). A value is well-formed if the size of its base value is less than or equal to the annotated public size bound.

Definition 1 (Well-formed value)**.** Value $(\!|v|\!)_z$ is well-formed, written $\vdash (\!|v|\!)_z$, if $size(v) \leq z$.

We say that store μ is well-formed with respect to typing environment Γ if the values bound in variables are well-formed and are of the correct type (Definition 2).

Definition 2 (Well-formed store w.r.t. typing environment)**.** Store μ is well-formed w.r.t. typing environment Γ, written $\Gamma \vdash \mu$, if $dom(\Gamma) = dom(\mu)$ and for all $x \in dom(\Gamma)$ we have

1) $\Gamma(x) = int@\ell \implies \exists n, z : \mu(x) = (\!|n|\!)_z \wedge size(n) \leq z$
2) $\Gamma(x) = string@\ell \implies \exists s, z : \mu(x) = (\!|s|\!)_z \wedge size(s) \leq z$

Well-formedness of memory m with respect to typing environment Δ is defined equivalently (Definition 3).

Definition 3 (Well-formed memory w.r.t. typing environment)**.** Store m is well-formed w.r.t. typing environment Δ, written $\Delta \vdash m$, if $dom(\Delta) = dom(m)$ and for all $x \in dom(\Delta)$ we have

1) $\Delta(x) = int@\ell \implies \exists n, z : m(x) = (\!|n|\!)_z \wedge size(n) \leq z$
2) $\Delta(x) = string@\ell \implies \exists s, z : m(x) = (\!|s|\!)_z \wedge size(s) \leq z$

We say that local environment π is well-formed with respect to typing environment Π if, for each channel, all values are well-formed and are of the correct type (Definition 4).

Definition 4 (Well-formed local message environment w.r.t typing environment)**.** Define local environment π to be well-formed with respect to typing environment Π, written $\Pi \vdash \pi$, if $dom(\Pi) = dom(\pi)$ and for all $ch \in dom(\Pi)$ such that $\Pi(ch) = \sigma@\ell$ we have $\vdash_\sigma \pi(ch)$ defined by the following rules:

$$\frac{}{\vdash_\sigma []} \qquad \frac{\vdash_\sigma tl}{\vdash_\sigma \bullet :: tl} \qquad \frac{size(n) \leq z \qquad \vdash_{int} tl}{\vdash_{int} (\!|n|\!)_z :: tl}$$

$$\frac{size(s) \leq z \qquad \vdash_{string} tl}{\vdash_{string} (\!|s|\!)_z :: tl}$$

Next, we define the potential q of trace τ with respect to typing environment Λ. Trace potential loosely bounds the number of future dummy messages that may result after observing a given trace. Genuine messages $\overleftrightarrow{ch}\ (t, 1, (\!|v|\!)_z)$ increase the potential of a trace by the annotated potential r of channel ch in Λ, while dummy messages $\overleftrightarrow{ch}\ (t, 0, (\!|v|\!)_z)$ decrease the potential by one. The definition does not capture that potential is strictly decreasing for all sends under non-public pc, but is nevertheless sufficient for showing our overhead theorem (Section V).

Definition 5 (Trace potential)**.** Define potential q of a trace τ with respect to typing environment Λ, written $\Lambda \vdash \tau : q$, by the following rules:

$$\frac{}{\Lambda \vdash \epsilon : 0} \qquad \frac{\Lambda \vdash \tau : q \qquad \Lambda(ch) = \_@\_;\_;r}{\Lambda \vdash \tau \cdot \overleftrightarrow{ch}\ (t, 1, (\!|v|\!)_z) : q + r}$$

$$\frac{\Lambda \vdash \tau : q + 1}{\Lambda \vdash \tau \cdot \overleftrightarrow{ch}\ (t, 0, (\!|v|\!)_z) : q}$$

Using Definition 5, we define well-formedness of network strategies ω with respect to typing environment Λ (Definition 6). A network strategy is well-formed if messages $ch(t, b, (\!|v|\!)_z)$ it produces are well-formed and of the correct type, and if dummy messages are only produced on non-public channels and only when the trace has sufficient potential for the dummy message and the annotated potential r of channel ch in Λ.

We enforce the bound by a well-formedness condition for network strategies ω (Definition 6) and prove that it is enforced by handlers in well-typed programs. We provide this proof in the accompanying technical report.

Definition 6 (Well-formed network strategy)**.** Network strategy ω is well-formed w.r.t. typing environment Λ, written $\Lambda \vdash \omega$, if for any τ such that $\omega(\tau) = ch(t, b, (\!|v|\!)_z)$ and $\Lambda(ch) = \sigma@\ell_{mode}; \_; r$ we have $size(v) \leq z$ and

- $\sigma = int \implies \exists n : v = n$
- $\sigma = string \implies \exists s : v = s$
- $b = 0 \implies \ell_{mode} \neq \bot \wedge \Lambda \vdash \tau : q + 1 + r$

We lift typing of programs, stores, local environments, and network strategies to typing of consumer states C, written $\Gamma, \Pi, \Lambda \vdash C$, in the straightforward way in Figure 9.

$$\frac{\Gamma, \Pi, \Lambda \vdash p \qquad \Gamma \vdash \mu \qquad \Pi \vdash \pi \qquad \Lambda \vdash \omega}{\Gamma, \Pi, \Lambda \vdash (p, \mu, \pi, \omega, h, \tau)}$$

Figure 9. Typing of system

IV. Noninterference

In this section define attacker knowledge, our security condition, and give our noninterference theorem that well-typed programs in *OblivIO* satisfy the security condition. We first define the equivalence relations necessary for our attacker knowledge definition and security condition.

We define stores μ_1, μ_2 to be equivalent up to level ℓ_{adv} with respect to typing environment Γ (Definition 7) if, for every variable, they agree on the values up to that level. We further require that the public sizes of values are the same.

Definition 7 (Equivalence of store up to level)**.** Stores μ_1 and μ_2 are equivalent up to level ℓ_{adv} w.r.t. typing environment Γ, written $\mu_1 \approx^{\Gamma}_{\ell_{adv}} \mu_2$, if for all $x \in dom(\Gamma)$ with $\Gamma(x) = \sigma@\ell$ we have that if $\mu_1(x) = (\!|v_1|\!)_{z_1}$ and $\mu_2(x) = (\!|v_2|\!)_{z_2}$ then

1) $z_1 = z_2$
2) $\ell \sqsubseteq \ell_{adv} \Rightarrow v_1 = v_2$

Equivalence of memories m_1, m_2 level ℓ_{adv} with respect to typing environment Δ is defined equivalently (Definition 8).

Definition 8 (Equivalence of memory up to level)**.** Memories m_1 and m_2 are equivalent up to level ℓ_{adv} w.r.t. typing environment Δ, written $m_1 \approx^{\Delta}_{\ell_{adv}} m_2$, if for all $x \in dom(\Delta)$ s.t. $\Delta(x) = \sigma@\ell$, we have that if $m_1(x) = (\!|v_1|\!)_{z_1}$ and $m_2(x) = (\!|v_2|\!)_{z_2}$ then

1) $z_1 = z_2$
2) $\ell \sqsubseteq \ell_{adv} \Rightarrow v_1 = v_2$

We define local environments π_1, π_2 to be equivalent up to level ℓ_{adv} with respect to typing environment Π, written $\pi_1 \approx^{\Pi}_{\ell_{adv}} \pi_2$, by equality on the streams at all channels up level ℓ_{adv} (Definition 9).

Definition 9 (Equivalence of local environment up to level)**.** Equivalence of local environments π_1 and π_2 up to level ℓ_{adv} w.r.t. typing environment Π, written $\pi_1 \approx^{\Pi}_{\ell_{adv}} \pi_2$, is defined by the following rule:

$$\frac{\Pi(ch) = \sigma@\ell \qquad \ell \sqsubseteq \ell_{adv} \implies \pi_1(ch) = \pi_2(ch)}{\pi_1 \approx^{\Pi}_{\ell_{adv}} \pi_2}$$

Two events α_1, α_2 are equivalent up to level ℓ_{adv} (Definition 10) if they agree on the properties observable at that level.

Definition 10 (Equivalence of output events up to level)**.** Equivalence of events α_1 and α_2, up to level ℓ_{adv} w.r.t. typing environment Λ, written $\alpha_1 \approx^{\Lambda}_{\ell_{adv}} \alpha_2$, is defined by as follows:

$$\frac{}{\epsilon \approx^{\Lambda}_{\ell_{adv}} \epsilon}$$

$$\frac{\Lambda(ch) = \sigma@\ell_{mode};\ell_{val};q \quad \ell_{mode} \sqsubseteq \ell_{adv} \implies b_1 = b_2 \quad \ell_{val} \sqsubseteq \ell_{adv} \implies v_1 = v_2}{\overset{\leftrightarrow}{ch}(t, b_1, (\!| v_1 |\!)_z) \approx^{\Lambda}_{\ell_{adv}} \overset{\leftrightarrow}{ch}(t, b_2, (\!| v_2 |\!)_z)}$$

We lift this definition to equivalence on traces, written $\tau_1 \approx^{\Lambda}_{\ell_{adv}} \tau_2$, in the straightforward way by point-wise equivalence (Definition 11).

Definition 11 (Trace equivalence up to level)**.** Equivalence of traces τ_1, τ_2 up to level ℓ_{adv} w.r.t. typing environment Λ, written $\tau_1 \approx^{\Lambda}_{\ell_{adv}} \tau_2$, is defined by the following rules

$$\frac{}{\epsilon \approx^{\Lambda}_{\ell_{adv}} \epsilon} \qquad \frac{\tau_1 \approx^{\Lambda}_{\ell_{adv}} \tau_2 \quad \alpha_1 \approx^{\Lambda}_{\ell_{adv}} \alpha_2}{\tau_1 \cdot \alpha_1 \approx^{\Lambda}_{\ell_{adv}} \tau_2 \cdot \alpha_2}$$

We define external event queues ω_1, ω_2 to be equivalent up to level ℓ_{adv} with respect to typing environment Λ (Definition 12) if they agree on the channel, timestamp and size of messages for equivalent traces, as well as message mode and value of messages on channels observable at that level.

Definition 12 (Equivalence of network strategies up to level)**.** Network strategies ω_1 and ω_2 are equivalent up to level ℓ_{adv} w.r.t. typing environment Λ, written $\omega_1 \approx^{\Lambda}_{\ell_{adv}} \omega_2$, if for any ch s.t. $\Lambda(ch) = \sigma@\ell_{mode};\ell_{val};q$ and any τ_1, τ_2 such that $\tau_1 \approx^{\Lambda}_{\ell_{adv}} \tau_2$ we have that if $\omega(\tau_1) = ch_1(t_1, b_1, (\!| v_1 |\!)_{z_1})$ and $\omega(\tau_2) = ch_2(t_2, b_2, (\!| v_2 |\!)_{z_2})$ then $ch_1 = ch_2$, $t_1 = t_2$, $z_1 = z_2$, and

- $\ell_{mode} \sqsubseteq \ell_{adv} \implies b_1 = b_2$
- $\ell_{val} \sqsubseteq \ell_{adv} \implies v_1 = v_2$

We lift the above equivalence definitions to define equivalence of consumer states, written $C_1 \approx^{\Gamma,\Pi,\Lambda}_{\ell_{adv}} C_2$, by straightforward lifting of the equivalences of the components (Definition 13).

Definition 13 (Equivalence of consumer states)**.** Define consumer states C_1, C_2 to be equivalent up to level ℓ_{adv}, with respect to typing environments Γ, Π, Λ, written $C_1 \approx^{\Gamma,\Pi,\Lambda}_{\ell_{adv}} C_2$, by the following rule:

$$\frac{\mu_1 \approx^{\Gamma}_{\ell_{adv}} \mu_2 \quad \pi_1 \approx^{\Pi}_{\ell_{adv}} \pi_2 \quad \omega_1 \approx^{\Lambda}_{\ell_{adv}} \omega_2 \quad \tau_1 \approx^{\Lambda}_{\ell_{adv}} \tau_2}{(p, \mu_1, \pi_1, \omega_1, h, \tau_1) \approx^{\Gamma,\Pi,\Lambda}_{\ell_{adv}} (p, \mu_2, \pi_2, \omega_2, h, \tau_2)}$$

To simplify our security condition, we define the projection of the trace component from configuration Q, written $trace(Q)$, in the straightforward way (Definition 14).

Definition 14 (Trace of configuration)**.** Define the projection of the trace of configuration Q, written $trace(Q)$, as follows:

$$trace(Q) = \begin{cases} \tau & \text{if } Q = (p, \mu, \pi, \omega, h, \tau) \\ \tau & \text{if } Q = (p, \overline{b}, c, m, \mu, \pi, \omega, h, \tau)^{ch} \end{cases}$$

We are now ready to define attacker knowledge (Definition 15). Given typing environments Γ, Π, Λ, a consumer state C, and a level ℓ_{adv}, we define attacker knowledge as the set of equivalent consumer states C' that produce an equivalent trace when run.

Definition 15 (Attacker knowledge)**.** Given consumer state C and trace τ such that $C \longrightarrow^* Q$, with $trace(Q) = \tau$, define attacker knowledge at level ℓ_{adv} as follows:

$$k_{\Gamma,\Pi,\Lambda}(C, \tau, \ell_{adv}) \triangleq \{C' \mid C \approx^{\Gamma,\Pi,\Lambda}_{\ell_{adv}} C' \wedge C' \longrightarrow^* Q' \wedge \tau \approx^{\Lambda}_{\ell_{adv}} trace(Q'_2)\}$$

We use the attacker knowledge definition to define our timing-sensitive, progress-sensitive security condition (Definition 16).

Definition 16 (Progress-sensitive noninterference)**.** Given consumer state C and a run $C \longrightarrow^* Q$ with $trace(Q) = \tau \cdot \alpha$, the run satisfies progress-sensitive noninterference if for all levels ℓ_{adv} we have $k_{\Gamma,\Pi,\Lambda}(C, \tau \cdot \alpha, \ell_{adv}) \supseteq k_{\Gamma,\Pi,\Lambda}(C, \tau, \ell_{adv})$.

With our security condition defined, we are ready to state the soundness theorem for our type system (Theorem 1).

Theorem 1 (Soundness)**.** *Given Γ, Π, Λ and consumer state C such that $\Gamma, \Pi, \Lambda \vdash C$. If $C \longrightarrow^* Q$ with $trace(Q) = \tau \cdot \alpha$ then $k_{\Gamma,\Pi,\Lambda}(C, \tau \cdot \alpha, \ell_{adv}) \supseteq k_{\Gamma,\Pi,\Lambda}(C, \tau, \ell_{adv})$.*

We provide the proof in the accompanying technical report.

V. Enforcement overhead

In this section we provide a bound on the traffic overhead introduced by *OblivIO*'s enforcement strategy. The bound is by a multiplicative factor given by typing environment Λ. We show this bound by considering two runs, one in the standard semantics given in Section II and one in an unsafe semantics that suppresses all dummy messages, and bounding the difference in the lengths of the traces (Theorem 2). Our model accounts for observing network traffic between other nodes by events $\widetilde{ch}(t, b, (\!| v |\!)_z)$ enabling us to bound the amount of dummy traffic across all network nodes.

The overhead in traffic produced by the enforcement can be categorised as either *direct* (dummy messages), or *indirect* (additional, genuine messages). As presented, *OblivIO* has no indirect overhead and produces the same genuine traffic under the standard and suppressing semantics. The language as presented contains no primitives that are affected by phantom computation, e.g., taking the size of values or taking the time. Such primitives could be introduced while still bounding direct overhead, but bounding indirect overhead would be difficult.

We first define the operational semantics of a system that suppressed dummy messages (Figure 10). Rules CC-Unsafe and CP-Unsafe require that network strategy ω produces messages with mode bit 1 to take a step, and rule PP-Unsafe filters all events that are not sends with mode bit 1.

Next, we define extension relations for relating values, traces, and network strategies up to the effects of phantom.

We define the extension on values as equality up to padding of the extended value (Definition 17).

Definition 17 (Extension of values)**.** Define value $(\!| v_2 |\!)_{z_2}$ to be an extension of value $(\!| v_1 |\!)_{z_1}$, written $(\!| v_1 |\!)_{z_1} \lessapprox (\!| v_2 |\!)_{z_2}$, if $v_1 = v_2$ and $z_1 \leq z_2$.

CC-Unsafe

$$\frac{\omega(\tau) = ch(t, 1, (\!|v|\!)_z) \qquad (p)(ch) \not\Downarrow}{(p, \mu, \pi, \omega, h, \tau) \xrightarrow[unsafe]{} (p, \mu, \pi, \omega, h, \tau \cdot \widetilde{ch}(t, 1, (\!|v|\!)_z))}$$

CP-Unsafe

$$\frac{\omega(\tau) = ch(t, 1, (\!|v|\!)_z) \quad (p)(ch) \Downarrow c, x \quad h' = h :: \mathsf{in}(ch, t, z) \quad \tau' = \tau \cdot \overleftarrow{ch}(t, 1, (\!|v|\!)_z)}{(p, \mu, \pi, \omega, h, \tau) \xrightarrow[unsafe]{} (p, [1], c, [x \mapsto (\!|v|\!)_z], \mu, \pi, \omega, h', \tau')^{ch}}$$

PP-Unsafe

$$\frac{\langle \overline{b}, c, m, \mu, \pi, h\rangle \xrightarrow{\alpha} \langle \overline{b}', c', m', \mu', \pi', h'\rangle \qquad \tau' = \begin{cases} \tau \cdot \alpha & \text{if } \alpha = \overleftarrow{ch}(t, 1, (\!|v|\!)_z) \\ \tau & \text{otherwise} \end{cases}}{(p, \overline{b}, c, m, \mu, \pi, \omega, h, \tau)^{ch} \xrightarrow[unsafe]{} (p, \overline{b}', c', m', \mu', \pi', \omega, h', \tau')^{ch}}$$

PC-Unsafe

$$\frac{}{(p, \overline{b}, \texttt{stop}, m, \mu, \pi, \omega, h, \tau)^{ch} \xrightarrow[unsafe]{} (p, \mu, \pi, \omega, h :: \mathsf{ret}, \tau)}$$

Figure 10. Operational semantics of suppressing system

We define trace τ_2 to be a phantom extension of trace τ_1 (Definition 18) if τ_1 contains no dummy messages and, for all genuine messages, the traces agree on channels, bits, and values up to value extension.

Definition 18 (Trace equivalence up to phantom)**.** Define trace τ_2 to be a phantom extension of trace τ_1, written $\tau_1 \dot{\lessapprox} \tau_2$, by the following rules:

$$\frac{}{\epsilon \dot{\lessapprox} \epsilon} \qquad \frac{\tau_1 \dot{\lessapprox} \tau_2}{\tau_1 \dot{\lessapprox} \tau_2 \cdot \overleftrightarrow{ch}\,(t, 0, (\!|v|\!)_z)}$$

$$\frac{\tau_1 \dot{\lessapprox} \tau_2 \qquad (\!|v_1|\!)_{z_1} \lessapprox (\!|v_2|\!)_{z_2}}{\tau_1 \cdot \overleftrightarrow{ch}\,(t, 1, (\!|v_1|\!)_{z_1}) \dot{\lessapprox} \tau_2 \cdot \overleftrightarrow{ch}\,(t', 1, (\!|v_2|\!)_{z_2})}$$

We define network strategy ω_2 to be a phantom extension of network strategy ω_1 with respect to typing environmtn Λ (Definition 19) if for related trace τ_1, τ_2 we have that if $\omega_2(\tau_2)$ returns a genuine message, then so does $\omega_1(\tau_1)$ and the values of the messages are related by extension.

Definition 19 (Network strategy)**.** Define network strategy ω_2 to be a phantom extension of network strategy ω_1 w.r.t typing environment Λ, written $\omega_1 \dot{\lessapprox}_\Lambda \omega_2$, by the following rule:

$$\frac{\tau_1 \dot{\lessapprox} \tau_2 \wedge \omega_2(\tau_2) = ch(t_2, 1, (\!|v_2|\!)_{z_2}) \implies \omega_1(\tau_1) = ch(t_1, 1, (\!|v_1|\!)_{z_1}) \wedge (\!|v_1|\!)_{z_1} \lessapprox (\!|v_2|\!)_{z_2}}{\omega_1 \dot{\lessapprox}_\Lambda \omega_2}$$

Finally, we define q^Λ_{max} as the maximum potential annotation of channels in typing environment Λ (Definition 20).

Definition 20 (Maximum potential)**.** Define maximum potential w.r.t Λ, written q^Λ_{max}, as follows

$$q^\Lambda_{max} = \max\{q \mid \_@\_; \_; q \in \mathrm{range}(\Lambda)\}$$

With the above definitions in place, we are ready to state our overhead theorem (Theorem 2). It says that given a configurations starting from the empty trace and considering a run in the suppressing semantics producing trace τ_1, then a run in the standard semantics with any extended network strategy, will produce an extended trace τ_2 that longer by at most a factor of q^Λ_{max}.

Theorem 2 (Overhead)**.** *Consider* $(p, \mu, \pi, \omega_1, h, \epsilon)$ *such that* $\Gamma, \Pi, \Lambda \vdash (p, \mu, \pi, \omega_1, h, \epsilon)$ *and* ω_2 *such that* $\omega_1 \dot{\lessapprox}_\Lambda \omega_2$*. If we have a run in the unsafe semantics* $(p, \mu, \pi, \omega_1, h, \epsilon) \xrightarrow[unsafe]{}{}^* Q_1$ *with* $trace(Q_1) = \tau_1$ *then we have a run in the safe semantics* $(p, \mu, \pi, \omega_2, h, \epsilon) \longrightarrow^* Q_2$ *with* $trace(Q_2) = \tau_2$ *such that* $\tau_1 \dot{\lessapprox} \tau_2$ *and* $|\tau_2| \leq |\tau_1| * (1 + q^\Lambda_{max})$*.*

The proof is provided in the accompanying technical report.

VI. Example: Auction service

In this section we demonstrate *OblivIO* by example. The example we shall use is a simple, secure auction service. Further examples can be found in the accompanying technical report.

We provide type annotations for handlers and variables with the convention that handler $\texttt{CH}_{\ell_{mode}}\ \$q\ (x : \sigma_{\ell_{val}})\ \{c\}$ at node NODE is a handler for channel NODE/CH such that

$$\Gamma, \Pi, \Lambda; [x \mapsto \sigma@\ell_{val}]; \ell_{mode} \vdash^q c$$

Potential q is zero where omitted.

The auction consists of users Alice and Bob, the auction house, and a simple timing service used by the auction house for timing rounds. We omit the code for Bob as it is equivalent to the code for Alice.

We consider users Alice and Bob to be trusted and only consider leaks to network level attackers. We therefore consider a simple two-point lattice $\{\mathtt{L}, \mathtt{H}\}$ with ordering $\mathtt{L} \sqsubseteq \mathtt{L}$, $\mathtt{L} \sqsubseteq \mathtt{H}$, and $\mathtt{H} \sqsubseteq \mathtt{H}$. All other flows are illegal.

```
1  ALICE // Node ID
2
3  var max_bid : int_H;
4
5  TO_LEAD_H $1 (bid : int_H) {
6    oblif bid <= max_bid
7    then send(AUCTIONHOUSE/ALICE_BID, bid);
8    else skip;
9  }
10
11 ...
```

Listing 2. Alice

Alice's client code is provided in Listing 2. Messages on channel ALICE/TO_LEAD inform her of what she must bid to take the lead. The channel is typed with non-public context label and can receive dummy messages in case Alice is already leading. When receiving a genuine message, Alice obliviously branches on whether the required bid is less than or equal to the maximum bid she is willing to make. If so, she makes the bid. When receiving a dummy message, the send on Line 7 will execute under phantom mode regardless of the value she receives. The channel is typed with potential 1 as it may produce a dummy message on channel AUCTIONHOUSE/ALICE_BID,

which is typed with potential 0. When the auction finishes, Alice is notified of the winner on channel AUCTIONHOUSE/AUCTION_OVER_NAME and the winning bid on channel AUCTIONHOUSE/AUCTION_OVER_BID.

```
1  AUCTIONTIMER // Node ID
2
3  var c : int_L;
4
5  BEGIN_L (i : int_L) {
6    c = i * 2000;
7    while (c > 0) do {
8      c = c - 1;
9    }
10   send(AUCTIONHOUSE/TICK, 0);
11 }
```

Listing 3. Auction timer

Listing 3 presents the code for the auction timer. It is a simple busy waiting loop that counts down to zero before sending a message on channel AUCTIONHOUSE/TICK.

```
1  AUCTIONSERVER // Node ID
2
3  var winner : string_H;
4  var winning_bid : int_H;
5  var round_counter : int_L;
6
7  ALICE_BID_H (bid: int_H) {
8    oblif winning_bid < bid
9    then {
10     winner ?= "Alice";
11     winning_bid ?= bid;
12   }
13   else skip;
14 }
15
16 TICK_L $4 (dmy: int_L) {
17   if round_counter > 0
18   then {
19     oblif winner != "Alice"
20     then send(ALICE/TO_LEAD, winning_bid+1);
21     else skip;
22     ...
23     round_counter = round_counter - 1;
24     send(AUCTIONTIMER/BEGIN, 1);
25   } else {
26     send(ALICE/AUCTION_OVER_NAME,winner);
27     send(ALICE/AUCTION_OVER_BID,winning_bid);
28     ...
29   }
30 }
```

Listing 4. Auction server

Listing 4 presents the auction server code. The server stores the current winner and the winning bid in variables with secret label H and a round counter with public label L. The server starts by sending messages to Alice and Bob and starts the auction timer. When receiving bids, the auction server checks whether the bid is greater than the current winning bid and if so updates winning bid and winner.

When receiving a message on channel TICK from the auction timer, the auction server checks how many rounds are remaining. As the number of rounds is public, the branching on Line 17 need not be oblivious. On Line 19, the server obliviously branches on whether Alice is the current leader and if not, it sends her a message informing her what she must bid to take the lead. Once the round counter reaches zero, the server informs Alice and Bob of the outcome of the auction.

VII. Implementation

In this section we demonstrate the practicality of *OblivIO* by developing an interpreter that implements the language semantics. The interpreter implements security critical operations as constant-time algorithms.

Best practice for constant-time programming is to write the code using low-level languages, as optimising compilers may change security critical code and void the constant-time property [27]. However, we argue that development of the interpreter acts as a sanity check on our semantics and ideas and demonstrates that they are, in principle, possible to implement in a low-level language. Our interpreter is written in OCaml, but contains no language specific features.

The interpreter represents integers in the straightforward way, assigning them fixed size. Strings are represented as as tuples `int*char array`, where the the character array contains the padded string value and the integer component denotes the length of the prefix of the character array corresponding to the actual string. This choice allows us to access indices in arrays based on their padded sizes. The interpreter supports arithmetic, boolean, and all comparison operations on integers. For strings, it supports comparisons = and !=, and concatenation ^. The interpreter gives integers fixed size hence $\oplus_{size}$ is defined trivially for all operations that return integer result. For concatenation of string $(\!|s_1|\!)_{z_1}, (\!|s_2|\!)_{z_2}$, we define $z_1 \hat{}_{size} z_2 = z_1 + z_2$.

In the algorithms presented below, we shall use notation $(\!|s|\!)_z$ for strings for consistency with the rest of the paper. The translation to the internal representation of the interpreter is straightforward. For simplicity, the presented algorithms assume integer representations of booleans and chars, leaving conversions implicit.

Algorithm 1 presents data-oblivious string comparison. The algorithm returns 1 if the strings are equal up to padding, otherwise 0. That is, SafeEq$((\!|s_1|\!)_{z_1}, (\!|s_2|\!)_{z_2}) = 1$ iff. $s_1 = s_2$. The algorithm stores whether any mismatch has been found in variable x, starting with comparing the secret lengths of the strings. The algorithm then checks for equality by taking the *xor* of the character values at every index i, masking out indices beyond the semantic strings by using bit b. This code is similar to code widely used in state of the art cryptographic libraries such as libsodium [16] and OpenSSL [17]. For simplicity, the presented algorithm assumes that the public lengths z_1, z_2 are equal. These lengths are public and we assume that padding by a known amount does not leak any secrets.

Our interpreter implements oblivious assignments using a deep-copy semantics and utilising algorithms for constant-time selection. We use a common algorithm for constant-time selection of integers i, j:

$$x = ((1 \; xor \; b) * i) \; lor \; (b * j)$$

Algorithm 1 Safe string comparison

Require: $z_1 = z_2$

function SAFEEQ($\langle\!| s_1 |\!\rangle_{z_1}, \langle\!| s_2 |\!\rangle_{z_2}$)
  $x \leftarrow size(s_1)\ lxor\ size(s_2)$
  $l \leftarrow z_1$
  for $i = 0, \ldots, l-1$ **do**
    $b \leftarrow i < min(size(s_1), size(s_2))$
    $x \leftarrow x\ lor\ (b\ land\ (s_1[i]\ lxor\ s_2[i]))$
  end for
  return $x = 0$
end function

From this, we derive an algorithm for constant-time selection of strings (Algorithm 2).

Algorithm 2 Safe string select

Require: $z_1 = z_2$

function SAFESELECT($b, \langle\!| s_1 |\!\rangle_{z_1}, \langle\!| s_2 |\!\rangle_{z_2}$)
  for $i = 0, \ldots, z_1 - 1$ **do**
    $s[i] \leftarrow ((1\ xor\ b) * s_1[i])\ lor\ (b * s_2[i])$
  end for
  return $\langle\!| s |\!\rangle_{z_1}$
end function

Function SAFESELECT takes three arguments: selection bit b, and strings $\langle\!| s_1 |\!\rangle_{z_1}$ and $\langle\!| s_2 |\!\rangle_{z_2}$. If $b = 0$ then $\langle\!| s_1 |\!\rangle_{z_1}$ is returned and if $b = 1$ then $\langle\!| s_2 |\!\rangle_{z_2}$ is returned. We again assume for simplicity that $z_1 = z_2$. The selection algorithm selects from either s_1 or s_2 by multiplying with a bit-mask computed from b. This kind of bit-masking is ubiquitous in constant-time programming.

Next, we present our algorithm for string concatenation (Algorithm 3).

Algorithm 3 Safe string concatenation

function SAFECONCAT($\langle\!| s_1 |\!\rangle_{z_1}, \langle\!| s_2 |\!\rangle_{z_2}$)
  $z \leftarrow z_1 + z_2$
  for $i = 0, \ldots, z-1$ **do**
    $s[i] \leftarrow 0$
    for $j = 0, \ldots, z_1 - 1$ **do**
      $c \leftarrow s_1[j]$
      $b \leftarrow (i = j)\ land\ (j < size(s_1))$
      $s[i] \leftarrow s[i]\ lor\ (b * c)$
    end for
    for $j = 0, \ldots, z_2 - 1$ **do**
      $c \leftarrow s_2[j]$
      $b \leftarrow (i = j + size(s_1))$
      $s[i] \leftarrow s[i]\ lor\ (b * c)$
    end for
  end for
  return $\langle\!| s |\!\rangle_z$
end function

Function SAFECONCAT takes strings $\langle\!| s_1 |\!\rangle_{z_1}, \langle\!| s_2 |\!\rangle_{z_2}$ as arguments and outputs $\langle\!| s_1 \hat{} s_2 |\!\rangle_{z_1+z_2}$, the concatenation of s_1, s_2 padded to size $z_1 + z_2$. This algorithm is our own invention and based on similar bit-masking as the algorithms above. To protect secret length $size(s_1)$, the algorithm iterates over both input strings for each index in the output, leading to $\mathcal{O}((z_1 + z_2)^2)$ time complexity.

VIII. Discussion

We have shown how *OblivIO* can be used for writing secure, reactive programs, and we have demonstrated the feasibility of our approach by developing an interpreter that implements security critical operations as constant-time algorithms.

The core language of *OblivIO* is simple yet expressive and the bit-stack approach for oblivious branching lends itself to data-oblivious methods such as bit-masking. However, every security critical operation needs a secure implementation. Each additional feature we introduce to the language needs a functionally correct, constant-time implementation. Implementing constant-time algorithms is arduous and non-trivial and is made even harder by optimising compilers, that force developers of constant-time code to fight the compiler or write complicated algorithms in low-level languages [27]. This state of affairs is clearly undesirable from a security perspective and stresses the need for secure compiler support that enable security properties of high-level code to be preserved during compilation.

A. Defences against traffic analysis

Traffic analysis attacks can broadly be divided into two categories: 1) *anonymity*, inferring *who* is participating in actions or communication, and 2) *confidentiality*, inferring *what* actions are performed or the contents of communication. Various approaches exist for defending against traffic analysis. However, approaches developed for protecting against one category of attacks are not necessarily suitable for protecting against the other. Concealing *who* is performing an action does not necessarily conceal *what* action is performed, and concealing *what* action is performed does not necessarily conceal *who* is participating.

OblivIO is designed for protecting confidentiality, protecting the *what* of interactions. We discuss existing approaches and their trade-offs with respect to protecting confidentiality in more detail. The trade-offs are summarised in Figure 11. We use ● to signify that the approach excels, ○ to signify that the approach performs poorly, and ◑ to signify that the approach falls somewhere in between.

1) System-level approaches: The majority of defences against traffic analysis have been developed at the system level [2] and rely on traffic morphing. Traffic morphing conceals genuine traffic by stretching out bursts of high activity and padding periods of low activity with dummy traffic. As the approach is program-agnostic, it is general and can in principle be applied to existing systems. This makes it highly permissive. The approach also incurs effectively no overhead in computation time as only genuine traffic needs to be processed. However, balancing traffic and latency overheads is challenging. In its simplest form, traffic morphing utilises constant rate padding, morphing and padding traffic to other nodes to a constant rate of fixed-size packets. Such constant-rate connections must be maintained for all other nodes the system observably sends

Approach	Traffic overhead	Latency overhead	Bandwidth overhead	CPU-time overhead	Permissiveness	Ease of use
System-level	○	○	◑	●	●	●
Constant-time	●	●	●	◑	○	○
Data-oblivious	◑	◑	◑	◑	◑	◑

Figure 11. Comparison of different approaches

traffic to. Setting a low rate introduces significant latency overheads as messages are buffered, while a high rate introduces significant, if not prohibitive, amounts of dummy traffic [3]. Due to this trade-off between bandwidth and latency overheads, traffic morphing is most suitable for systems that either produce relatively constant rates of traffic to a fixed set of other nodes, or systems that can tolerate long network delays [19].

While traffic morphing is in principle general and permissive, Cherubin et al. [2] note that the approach may require substantial changes to applications and network stack, making deployment unfeasible in practice.

2) Program-level approaches: Compared to system-level approaches, program-level approaches for mitigating traffic analysis are less explored. The program-level approach enforces security by imposing restrictions on the program. This makes the approach less general and less permissive, but at potentially reduced overheads compared to system-level approaches. Program-level approaches can ensure that programs are safe by construction, and as all padding (if any) is performed at the level of the program, no changes are required to the network stack, as at all traffic is genuine at the level of the network.

a) Constant-time programming: A simple, but restrictive, paradigm for eliminating timing leaks is constant-time programming. The approach restricts the use of secret data input for a number of standard language features such as branching, indexing arrays, and variable-time operations like division. To get around these restrictions, constant-time programs inline computation from conditional branches and make clever use of bit-masking to ensure correctness of the program. Inlining branches incurs some overhead in computation time, but the approach introduces no direct overhead in traffic as no commands are conditionally executed depending on secrets. If a system can be feasibly rewritten as constant-time programs this approach is preferential. However, as Almeida et al. [9] note, adhering to constant-time programming is hard and requires developers to deviate from conventional programming practices. This makes the approach suitable only for simple programs.

b) Data-oblivious, reactive programming: Data-oblivious programming eases the restrictions imposed by constant-time programming by providing high-level support for writing constant-time code, thereby also easing the burden on developers. The principal idea of data-oblivious languages is oblivious conditionals. Oblivious conditionals execute both branches while negating unwanted side-effects of the non-chosen branch. This ensures that control flow and memory accesses are independent of secrets. *OblivIO* extends this principle to the reactive setting, where programs receive and react to network messages. Our approach incurs an overhead in traffic by introducing dummy messages for send commands that are conditionally executed depending on secrets. This overhead is bounded by a multiplicative factor that is statically known from typing. Our approach incurs a bandwidth overhead as the size of message values is padded to a public upper bound. Bounding the bandwidth overhead would require further restrictions to the language. Our approach introduces an overhead in computation time by executing both branches of oblivious conditionals, but we restrict the language to ensure termination of non-chosen branches. We do not provide a bound on the overhead in latency, but note that the number of dummy messages is bounded and handling them is guaranteed to terminate, hence we are guaranteed to respond to genuine messages eventually. Compared to system-level approaches, our approach is particularly suitable for programs that do not produce constant rates of traffic to predetermined parties. We discuss the main limitations of our approach below.

B. Local channels

Our model assumes that messages on local channels may be sampled in constant-time. This assumption is standard in the literature (e.g. [26]). Incoming messages must be processed and prepared for the running application in shared memory, yet at the same time must not have any effect on the execution time of the application. This problem arises from the need to protect message presence. To fully achieve this, fixed-rate scheduling could be used for a separate thread, dedicated to process and prepare local input.

C. Limitations

Programs in *OblivIO* are static and functions are not first class. This simplifies our model and helps more clearly deliver the core concepts of our approach. However, dynamic features in reactive programs are becoming more and more prevalent, albeit at the cost of introducing leaks [1], and cloud computing sees code being sent to remote servers to be run. Conceptually, the principles we have developed for *OblivIO* could be extended to handle public, dynamic code. Secret code entails secret control flow, which goes against the core design principle of the language. While it is straightforward to protect which of finitely many pieces of code is genuinely executed (by simply executing all but one in phantom mode) it is not clear how we could protect executing arbitrary code without severe restrictions. The data-oblivious comparison and selection algorithms we present in Section VII can be extended to new data structures built from the value types we consider, such as pairs or lists. Here, functions types are again different. It is not clear how constant-time selection and padding could be extended to functions.

Our model assumes that it is public which handler (if any) is triggered by a network message. It is conceptually

straightforward to hide this up to some degree, though at the cost of overhead in execution time and traffic, e.g., by setting up anonymity groups such that receiving an event for one handler would trigger the execution of all handlers in the group, with only the genuine handler executing in real mode.

Dynamically registering new secret handlers appears difficult. If a handler for a channel is registered in phantom mode, how should an incoming network message on that channel be handled? We could of course run the handler in phantom mode, but as the handler could produce network traffic this does not appear desirable in general. If multiple handlers are potentially registered depending on secrets, all handlers would need to be run in order to protect the secrets.

Channels are not first class in our language. There is again no great conceptual difficulty in supporting standard operations on channels if it is not secret which channels are operated on. String identifiers for nodes and handlers could provide an intuitive model for channels, enabling padding and conditional assignment, but sending on conditionally bound channels would reveal which channel is bound as shown by the following example, where an attacker can infer the secret by observing whether the message goes to `ALICE` or `BOB`:

```
1 oblif secret
2 then ch = ALICE/GREET
3 else ch = BOB/GREET;
4 send(ch,"Hello");
```

It may appear that an anonymous communication channel, possibly connecting via a set of relays, could be used to hide whom a message is sent to and thereby protect the secret. Unfortunately, it is not enough to send a single message for this program if the attacker knows the programs running at `ALICE` and `BOB`. Suppose `ALICE` replies to messages while `BOB` does not. An attacker armed with this knowledge could infer which node was sent to based on whether a reply is sent.

Our model does not allow real mode computation when handling dummy messages and does not permit genuine messages to be sent in response to dummy messages. Redefining the semantic rules to lift these restrictions would be straightforward, but would complicate reasoning about program correctness for developers and would make it difficult to bound the overhead in traffic.

IX. Related work

Previous work has applied language-based methods to the reactive programming model. Bohannon et al. [13] develop a security type system enforcing noninterference for reactive programs. However, covert channels, such as timing channels, are outside of the model they consider.

A. Secure multi execution and faceted values

Another approach, which has gained traction in practice, is Secure Multi Execution (SME) [28]. As the name suggests, SME executes a code snippet multiple times – once per security level – carefully restricting the in- and output of each execution to only the legal channels. The promise of SME is that secure programs are not adversely affected by the mechanism, a property called *transparency*. Coupled with a scheduler that prioritises low-execution this approach can be applied in a black-box fashion to programs and protects against timing leaks. However, the low-priority scheduling discipline faces issues in the reactive setting as it does not extend to executing handlers for multiple events [29]. Instead, reactive systems incorporating the SME principles such as FlowFox, a Firefox extension for secure information flow in JavaScript, settles for low-priority scheduling on a per-event basis [30]. This compromise unfortunately introduces a leak, as high-execution of a handler for one event taints the low-execution of the handler for the next.

Rafnsson and Sabelfeld [29] develop a fine-grained version of SME that models a network level attacker that observes when messages are sent. They employ a scheduler that enforces that low-runs never "outrun" high-runs to synchronise I/O between the runs. Their transparency property states that timing-sensitive noninterfering programs are not adversely modified in their I/O behaviour. Almeida et al. [9] point out that adhering to constant-time programming requires expertise and forces developers to deviate from conventional programming practices.

Execution using faceted values [31] has many of the same advantages and disadvantages as SME. While conditional assignment may intuitively seem related to faceted values, the above considerations for timing-channels in reactive programs under SME also apply here and make faceted values unsuitable. Our approach uses single, real values, regardless of execution mode, simplifying constant-time algorithms used for operations. Furthermore, our approach has the advantage of ruling out insecure programs through typing, rather than being limited to transparency for secure programs.

B. Timing channels and reactive programs

Bastys et al. [5] point out that remote attackers do not in general know when a program is started. They develop Clockwork, a monitor that rules out timing leaks in batch programs where starting time is not observed. They demonstrate the permissiveness of their model with the program `if` h `then` h_1 `:=` h_2`; send(`L`,1)`, which is considered unsafe in many models, and indeed would not be accepted by the type system of *OblivIO*. However, without knowledge of when the program starts the program is safe. This observation does not easily extend to the reactive programs and the network level attackers we consider, simply because it is not possible to react to network messages that have not yet been received. In our model, it is known when an event arrives at a node, and failure to produce any expected responses is observable hence an attacker can easily infer that a system is not running.

Blaabjerg and Askarov [20] consider program-level mitigation of traffic analysis by separating traffic shape from traffic content. Their approach does not mitigate timing differences from sensitive conditionals hence secure programs must set up schedules for all future traffic before executing such conditionals. The approach therefore does not easily extend to the reactive setting.

McCall et al. [32] consider how dynamic features in reactive programs, such as registering new handlers, can be used to leak information by abusing declassification policies. They design new SME rules to enforce separation of the declassification module from dynamically generated components.

Vassena et al. [33] propose a dynamic information flow control parallel runtime system that supports deterministic parallel thread execution. Such methods could be used in *OblivIO* to support the parallel processing needed for handling input on local channels.

C. Constant-time execution

As discussed in Section VIII, writing constant-time algorithms in a high-level language is not in general sufficient. Optimising compilers may rewrite the code, introduce branches, and otherwise break the constant-time guarantees. Barthe et al. [34] consider the problem of preserving side-channel countermeasures, such as constant-time code, during compilation, and present a framework for proving that a compilation pass preserves such countermeasures.

Cauligi et al. [35] present a DSL for writing constant-time cryptographic code and compiling it to LLVM bitcode. Their DSL allows programmers to write familiar, high-level code with variables annotated by security labels. Their compiler uses the security labels to transform unsafe behaviour that depends on secrets into constant-time code. They use the dudect analysis tool [36] to check that machine code compiled from their generated LLVM bitcode is constant-time.

Dantas et al. [37] investigate timing analysis countermeasures in the presence of just-in-time compilation, such as in the JavaVM. Their empirical results indicate that static countermeasures fare worse in this setting, while dynamic countermeasures retain much of their effectiveness.

D. Resource-awareness

Hofmann and Jost [21] introduce linear potentials for analysing the resource consumption of programs with linear bounds. Hoffmann and Hofmann [22] extend the notion of potentials to polynomial potentials, allowing analysis of programs with polynomial bounds on resource consumption. Krishnaswami et al. [38] explores the notion of potentials in functional, reactive programming and present a language that statically bounds the size of the data-flow graph of reactive programs. Dehesa-Azuara et al. [39] explore a notion of resource-aware noninterference in a setting without IO, where the sizes of program values are known to the attacker.

X. Conclusion

We consider the problem of mitigating traffic analysis attacks against online services and applications written in the reactive programming model. We show that data-oblivious computation is a natural fit for preventing timing-channels in this setting when coupled with information-rich dummy traffic. We develop the language *OblivIO*, a language for data-oblivious, reactive programs. We prove that well-typed programs in the language are secure against traffic analysis by convincingly padding traffic with dummy messages, and we show that the overhead introduced by our approach is bounded. We demonstrate the expressiveness of our language by example and the practicality of the language by developing an interpreter that implements security critical operations as constant-time algorithms.

XI. Acknowledgements

We thank the anonymous reviewers for their valuable suggestions for improving the paper. This work was funded by the Danish Council Independent Research for the Natural Sciences (DFF/FNU, project 6108-00363).

References

[1] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in web applications: A reality today, a challenge tomorrow," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 191–206.

[2] G. Cherubin, J. Hayes, and M. Juárez, "Website fingerprinting defenses at the application layer." *Proc. Priv. Enhancing Technol.*, vol. 2017, no. 2, pp. 186–203, 2017.

[3] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton, "Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail," in *2012 IEEE symposium on security and privacy*. IEEE, 2012, pp. 332–346.

[4] A. Askarov, D. Zhang, and A. C. Myers, "Predictive black-box mitigation of timing channels," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010, pp. 297–307.

[5] I. Bastys, M. Balliu, T. Rezk, and A. Sabelfeld, "Clockwork: Tracking remote timing attacks," in *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*. IEEE, 2020, pp. 350–365.

[6] A. Russo, J. Hughes, D. Naumann, and A. Sabelfeld, "Closing internal timing channels by transformation," in *Annual Asian Computing Science Conference*. Springer, 2006, pp. 120–135.

[7] D. Zhang, A. Askarov, and A. C. Myers, "Language-based control and mitigation of timing channels," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2012, pp. 99–110.

[8] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe, "Constant-time foundations for the new spectre era," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 913–926. [Online]. Available: https://doi.org/10.1145/3385412.3385970

[9] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 53–70.

[10] S. Zahur and D. Evans, "Obliv-c: A language for extensible data-oblivious computation," *IACR Cryptol. ePrint Arch.*, p. 1153, 2015. [Online]. Available: http://eprint.iacr.org/2015/1153

[11] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi, "Oblivm: A programming framework for secure computation," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 359–376.

[12] T. Nurkiewicz and B. Christensen, *Reactive Programming with RxJava: Creating Asynchronous, Event-based Applications*. " O'Reilly Media, Inc.", 2016.

[13] A. Bohannon, B. C. Pierce, V. Sjöberg, S. Weirich, and S. Zdancewic, "Reactive noninterference," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 79–90.

[14] S. Moore and S. Chong, "Static analysis for efficient hybrid information-flow control," in *2011 IEEE 24th Computer Security Foundations Symposium*. IEEE, 2011, pp. 146–160.

[15] A. Askarov, S. Chong, and H. Mantel, "Hybrid monitors for concurrent noninterference," in *2015 IEEE 28th Computer Security Foundations Symposium*. IEEE, 2015, pp. 137–151.

[16] F. Denis, "libsodium," 2021, accessed: 2021-10-01. [Online]. Available: https://github.com/jedisct1/libsodium

[17] OpenSSL, "Openssl," 2021, accessed: 2021-10-01. [Online]. Available: https://github.com/openssl/openssl

[18] C. Diaz, S. J. Murdoch, and C. Troncoso, "Impact of network topology on anonymity and overhead in low-latency anonymity networks," in *International Symposium on Privacy Enhancing Technologies Symposium*. Springer, 2010, pp. 184–201.

[19] N. Apthorpe, D. Y. Huang, D. Reisman, A. Narayanan, and N. Feamster, "Keeping the smart home private with smart (er) iot traffic shaping," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 3, 2019.

[20] J. F. Blaabjerg and A. Askarov, "Towards language-based mitigation of traffic analysis attacks," in *2021 IEEE 34th Computer Security Foundations Symposium (CSF)*. IEEE, 2021, pp. 1–15.

[21] M. Hofmann and S. Jost, "Static prediction of heap space usage for first-order functional programs," *ACM SIGPLAN Notices*, vol. 38, no. 1, pp. 185–197, 2003.

[22] J. Hoffmann and M. Hofmann, "Amortized resource analysis with polynomial potential," in *European Symposium on Programming*. Springer, 2010, pp. 287–306.

[23] J. Hoffmann, K. Aehlig, and M. Hofmann, "Resource aware ml," in *International Conference on Computer Aided Verification*. Springer, 2012, pp. 781–786.

[24] D. Clark and S. Hunt, "Non-interference for deterministic interactive programs," in *International Workshop on Formal Aspects in Security and Trust*. Springer, 2008, pp. 50–66.

[25] D. Hedin and D. Sands, "Timing aware information flow security for a javacard-like bytecode," *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 1, pp. 163–182, 2005.

[26] A. Sabelfeld and H. Mantel, "Static confidentiality enforcement for distributed programs," in *International Static Analysis Symposium*. Springer, 2002, pp. 376–394.

[27] L. Simon, D. Chisnall, and R. Anderson, "What you get is what you c: Controlling side effects in mainstream c compilers," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2018, pp. 1–15.

[28] D. Devriese and F. Piessens, "Noninterference through secure multi-execution," in *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 109–124.

[29] W. Rafnsson and A. Sabelfeld, "Secure multi-execution: Fine-grained, declassification-aware, and transparent," *Journal of Computer Security*, vol. 24, no. 1, pp. 39–90, 2016.

[30] W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens, "Flowfox: a web browser with flexible and precise information flow control," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 748–759.

[31] T. H. Austin and C. Flanagan, "Multiple facets for dynamic information flow," in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2012, pp. 165–178.

[32] M. McCall, H. Zhang, and L. Jia, "Knowledge-based security of dynamic secrets for reactive programs," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 175–188.

[33] M. Vassena, G. Soeller, P. Amidon, M. Chan, J. Renner, and D. Stefan, "Foundations for parallel information flow control runtime systems." in *POST*, 2019, pp. 1–28.

[34] G. Barthe, B. Grégoire, and V. Laporte, "Secure compilation of side-channel countermeasures: the case of cryptographic "constant-time"," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*. IEEE, 2018, pp. 328–343.

[35] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan, "Fact: a dsl for timing-sensitive computation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 174–189.

[36] O. Reparaz, J. Balasch, and I. Verbauwhede, "Dude, is my code constant time?" in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1697–1702.

[37] Y. G. Dantas, T. Hamann, and H. Mantel, "A comparative study across static and dynamic side-channel countermeasures," in *International Symposium on Foundations and Practice of Security*. Springer, 2018, pp. 173–189.

[38] N. R. Krishnaswami, N. Benton, and J. Hoffmann, "Higher-order functional reactive programming in bounded space," *ACM SIGPLAN Notices*, vol. 47, no. 1, pp. 45–58, 2012.

[39] M. Dehesa-Azuara, M. Fredrikson, J. Hoffmann *et al.*, "Verifying and synthesizing constant-resource implementations with types," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 710–728.