PrivacyScope: Automatic Analysis of Private Data Leakage in TEE-Protected Applications

Ruide Zhang[1], Ning Zhang[2], Assad Moini[1], Wenjing Lou[1], and Y. Thomas Hou[1]

[1]Virginia Polytechnic Institute and State University, Blacksburg, VA
[2]Washington University, St. Louis, MO

***Abstract*—Big data analytics is having a profound impact on many sectors of the economy by transforming raw data into actionable intelligence. However, increased use of sensitive business and private personal data with no or limited privacy safeguards has raised great concerns among individuals and government regulators. To address the growing tension between the need for data utility and the demand for data privacy, trusted execution environment (TEE) is being used in academic research as well as industrial application as a powerful primitive to enable confidential computation on the private data with only the result disclosed but not the original private data. While much of the current research has been focusing on protecting the TEE against attacks (e.g. side-channel information leakage), the security and privacy of the applications executing inside a TEE enclave has received little attention. The general attitude is that the application is running inside a trusted computing base (TCB), and therefore can be trusted. This assumption may not be valid when it comes to unverified third-party applications.**

In this paper, we present PrivacyScope, a static code analyzer designed to detect leakage of private data by an application code running in a TEE. PrivacyScope accomplishes this by analyzing the application code and identifying violations of a property called nonreversibility. We introduce nonreversibility since the classical noninterference property falls short of detecting private data leakage in certain scenarios, e.g., in machine learning (ML) programs where the program output is always related to (private) input data. Given its strict reliance on observable state, the noninterference falls short of detecting private data leakage in these situations. By design, PrivacyScope detects both explicit and implicit information leakage. The nonreversibility property is formally defined based on the noninterference property. Additionally, we describe the algorithms for PrivacyScope as extensions to the runtime semantics of a general language. To evaluate the efficacy of our approach and proof-of-feasibility prototype, we apply PrivacyScope to detect data leakage in select open-source ML code modules including linear regression, k-means clustering and collaborative filtering. Also, PrivacyScope can detect intentional data leakage code injected by a programmer. We responsibly disclosed all the discovered vulnerabilities leading to disclosure of private data in the open-source ML program we analyzed.

***Index Terms*—information flow analysis, privacy, intel sgx enclave, static analysis, symbolic execution, taint analysis, formal semantics, machine learning, internet of things**

I. Introduction

With more and more data being collected and analyzed, there is an increasing concern on privacy implication of the sensitivity of information on individuals. While we are enjoying such rapid advancement in data science, many consider this a step backwards on the fundamental civil right to privacy. In an effort to tackle this fundamental trade-off between data utility and data privacy, much work has been done to enable secure computation on confidential data, where only the results are revealed but not the original data. Secure computation techniques are generally divided into two categories, cryptographic techniques [1] and system mechanisms [2]–[4]. Trusted Execution Environment (TEE) is one of the more popular system methods designed to ensure secure computation on private data given its ability to host arbitrary computation with limited overhead. However, even though techniques leveraging TEE aim at providing privacy assurance to users, the security protection of system actually depends on both the TEE and the TEE-protected applications themselves. For example, while the Intel SGX architecture can guarantee the integrity and confidentiality of execution, it does not address leakage of private data due to program code vulnerabilities or intentionally injected backdoors within SGX-protected program. The code executing in a SGX enclave can inadvertently or maliciously leak private data outside its trust boundary. Majority of recent research [5]–[7] focus on addressing potential data leakage on TEE, while few works [8] focus on private data leakage by TEE-protected applications themselves.

Since TEE-protected applications may contain malicious logic embedded by attacker or data leakage bugs brought by programmers, it is important for users of these secure applications to audit and validate them. However, ensuring trustworthiness of TEE-protected applications manually requires security expertise and is not scalable for upcoming large amount of TEE-protected applications. Therefore, an automatic verification tool for users to detect leakage in TEE-protected applications is desired. [8] formally specifies semantics of TEE-protected application and applies classical information flow analysis [9] to automatically detect leakage on it. The classical information flow analysis applies the noninterference property that essentially ensures no mutation of sensitive data can lead to observable changes in program state from the perspective of an outside observer. However, the noninterference property is not suitable for widely adopted ML algorithms in the era of IoT and cloud computing. ML algorithms use private data to train models which are observable for cloud service provider. Thus, ML programs always violate classical noninterference property. Therefore, a new property is desired for defining information leakage in

ML programs.

Inspired by the noninterference property, we formally define nonreversibility property of enclave application in this paper. Violations of nonreversibility property implies malicious actor can infer sensitive input by observing the output generated by program code running in an enclave. Thus, nonreversibility is applicable to analyzing data leakage issues in widely adopted ML code modules. Detecting violation of nonreversibility is challenging, there are fundamental challenges that are different from noninterference. First, it is not trivial to determine if the recovery from output to input is deterministic or not. Second, even if there is a data flow from sensitive input to output, it is not entirely clear what the exact relationship is between input and output, which is crucial in determining recoverability. To tackle these challenges, we design and develop PrivacyScope, a static code analyzer that detects violations of the nonreversibility property by enclave program code. PrivacyScope employs symbolic execution to track propagation of private data and records path conditions when program branches throughout the exploration based on a symbolic program input. At the conclusion of program analysis, PrivacyScope generates a report detailing any leakage of private data. PrivacyScope works seamlessly with the secure development environment. As a demonstration, we extend the Intel SGX software development ecosystem with PrivacyScope to automatically detect violation of nonreversibility property on enclave modules.

The main contributions of this work are as following:

- We formally define *nonreversibility* property to characterize the notion of secret data leakage in ML programs. Inspired by noninterference property, nonreversibility accomplishes this by establishing a deterministic relationship between program input and output in TEE-protected application.
- We construct PRIML language to formally describe our proposed innovative approach, PrivacyScope, which automatically detects violations of the nonreversibility property in a TEE-protected application.
- We present a proof-of-feasibility implementation of PrivacyScope leveraging the Clang Static Analyzer and demonstrate the viability and efficacy of our approach by evaluating its performance by analyzing select ML applications executing in Intel SGX enclaves.

II. Background

A. Intel SGX

Intel Software Guard Extensions (SGX) is an extension of the Intel's processor architecture designed to safeguard code and data against unauthorized modification and disclosure. SGX guarantees the integrity and confidentiality of user code and data by providing a processor-hardened, processor-protected, trusted execution environment called an enclave. A user application executing within an enclave is subject to heightened security measures enforced by the processor. Remote attestation, key and credential provisioning are other critical features of the Intel SGX architecture. Remote attestation allows a remote party to verify the authenticity of application code module executing inside an enclave.

Despite its strength, SGX suffers from several hardware security limitations including SGX page faults, cache timing, address bus monitoring and processor monitoring [5]. There has been research working on addressing these limitations like defending against cache timing attacks in [10]. We consider these security limitations orthogonal to our intent of this paper and believe these limitations must be addressed independently from PrivacyScope.

B. Symbolic Execution

Symbolic execution is a popular program analysis technique that dates back to the 1970s to test whether certain properties can be violated by a piece of software [11], [12]. The key idea is to allow a program to take on symbolic inputs. Then the program is abstractly interpreted by an symbolic execution engine. During interpretation, path condition and symbolic memory store are recorded for each explored control flow path.

By symbolically interpreting TEE-protected applications, PrivacyScope logs symbolic expression of targeted input arguments and uses logged information to track any explicit leakages. Additionally, by tracking target input arguments in path conditions and by combining that information with the returned result from the application, PrivacyScope can detect any implicit leakages. An alternative way to find explicit leakage is to use data flow analysis (DFA) frameworks [13], [14]. Symbolic execution is orders more complex in terms of complexity comparing to DFA. However, most data flow frameworks are path insensitive and are hard to be used for finding implicit leakages.

C. Clang Static Analyzer

The Clang Static Analyzer is an open source analysis tool for finding bugs in C/C++, and Objective-C programs during compilation phase. The analyzer is a symbolic execution engine that abstractly interprets program code and traces out possible execution paths. After generating the possible execution paths, the analyzer performs conceptually a reachability analysis. During the analysis, the analyzer enters predefined bug checkers. A bug is found by hitting a state where some violation of checking invariants are satisfied. PrivacyScope is a special checker we create, which is implemented on top of Clang Static Analyzer for identifying explicit and implicit leakage of enclave program.

Clang Static Analyzer leverages a region-based memory model to perform path-sensitive symbolic program analysis [15]. The Analyzer can process most forms of C expressions, containing arbitrary levels of pointer dereferencings, pointer arithmetics, composite arrays and struct data types, arbitrary type casts, dynamic memory allocations, etc. These features are critical to PrivacyScope, given that pointer operation is commonplace in C/C++ code and composite structs are widely used in all C/C++ software including data analytic modules.

III. Threat Model And Assumptions

The goal of PrivacyScope is to discover deterministic leakage of user private data of an ML application. The threat model follows that of TEE-based secure computation. User private data are encrypted for storage outside the environment, and when they are used for training the mode, the data is decrypted only inside the container for consumption. With recent advances in machine learning, along with the packaging, it is becoming increasingly accessible to the general public, even to those without a machine learning background. As privacy concerns continue to grow, it is likely that the majority of the computation on user data will be conducted in a privacy-preserving manner. And the security of the applications running in the container, which are granted unlimited access to user private data, will be crucial in user privacy protection. However, with new customizations of individual training methods for various application domains, it can be very challenging for an individual user of the ML system, potentially without any expertise on programming language, information flow and machine learning to recognize subtle ways the ML applications can deterministically leak user data. As a result, we assume that there can be unintentional or intentional logic in the TEE-protected application that will leak contents deterministically. PrivacyScope is a detection framework that can take user-defined privacy leakage rules written as PRIML language extension detailed in Section V-A, and automatically analyze TEE-protected programs to see if there is any deterministic information flowing from the input (such as user private data) to the output (such as ML models).

Although PrivacyScope aims at preventing the TEE-protected application from leaking user's private data in a deterministic manner. It is not designed to detect potential information leakage due to various side channels or covert channels. We provide a brief discussion in Section VIII-A on potential extension to protect against side/covert channel information leakage.

IV. Nonreversibility Property

In traditional definition of secret leakage, for a user to keep her data confidential, she would define a policy stating that change of her confidential data cannot affect any publicly observable data. This policy allows programs to perform manipulation and modification on secret data, as long as any observable outputs of these programs do not reveal information about the secret data. Since this policy states that no visible public data is interfered with confidential data, this sort of policy is called a noninterference policy [16]. Intuitively, noninterference for programs guarantees that "*a variation of confidential (high) inputs does not cause a variation of public (low) outputs*" [17].

However, noninterference policy is not suitable for our scenario. In our scenario, machine learning algorithm sits in enclave. We view secret inputs received by enclave as high and the training output to the cloud server as low. In traditional definition, any change of high data would not interfere with low data. But output trained model changes according to received input secret data. In this case, noninterference policy is always violated in machine learning algorithm. Thus, we need a finer grade policy. In contrast to noninterference, we define nonreversibility to guarantee that *a variation of a single confidential (high) input could cause a variation of public (low) outputs, but keeping this single confidential (high) input and (low) inputs unchange will not always lead to the same public (low) outputs*. We provide the formalization of nonreversibility in the following.

We extend notations from [17] and rigorously formalize noninterference and nonreversibility using the machinery of programming-language semantics. We assume that computation starts in an input state $s = (s_H, s_L)$. s_H and s_L contain the initial values of variables of $high$ and low, respectively. s_h represents any single variable inside s_H and s_l represents any single variable inside s_L. The program either terminates in an output state $s' = (s'_H, s'_L)$ with output values for the high and low variables, or diverges. Thus, the semantics $[\![P]\!]$ of a program P is a function $[\![P]\!] : S \to S_\perp$ (where $S_\perp = S \cup \perp$ and $\perp \notin S$) which maps an input state $s \in S$ either to an output state $[\![P]\!]s \in S$, or to $\perp$ if the program fails to terminate. We define *equivalence relations* $=_L$ and $=_h$. $=_L$ means two states are the same if they are equal on all the low variable (i.e. $s =_L s'$ if and only if $\forall s_l \in s_L$ and $\forall s'_l \in s'_L, s_l = s'_l$). $=_h$ means there exists one variable inside the high variables are the same for two states (i.e. $s =_h s'$ if and only if $s_h = s'_h$). We characterize the observation power of an attacker by a relation $\approx_L$ on behaviors such that two behaviors are related by $\approx_L$ if and only if they are indistinguishable to the attacker. Relation $\approx_L$ implies that the attacker can observe the low variables. For a given semantic model, noninferference is formalized as follows. P is secure if and only if $\forall s_1, s_2 \in S, s_1 =_L s_2 \implies [\![P]\!]s_1 \approx_L [\![P]\!]s_2$. This reads "if two input states share the same low values, then the behaviors of the program executed on these states are indistinguishable by the attacker." Whereas, nonreversibility is formalized as below. P is secure if and only if

$$\forall s_1, s_2 \in S$$

$$s_1 =_L s_2$$

$$s_1 =_h s_2$$

$$\exists h' \neq h, s_{1_{h'}} \neq s_{2_{h'}} \implies [\![P]\!]s_1 \not\approx_L [\![P]\!]s_2$$

which reads "if two input states share the same low values and the same value of one single high variable, then there exists one other high variable that when it is different for the two input states, attacker will observe different behaviors of the program executed on these states." If this other high variable does not exist, then one single high variable will be leaked and P will not be secure. Because, an attacker would be able to look at P and reverse the related computations of that single high variable. According to the formal definition of nonreversibility, the program $l := h_1 + 4$ is insecure and the value of h_1 can be inferred by attacker by observing l.

However, the program $l := h_1 + 4 + h_2$ is secure because if h_2 is changed, l observed by attacker will also be changed. And attacker cannot infer value of h_1 without knowledge of h_2. Note that, the probability distribution of inferring h_1 from l will thus be determined by h_2 in this case.

V. PRIVACYSCOPE

A. A General Language: PRIML

For precise declaration of how PrivacyScope works under the hood, we extend notations in [18] and introduce a language called PRIML: PRivacyscope InterMediate Language. PRIML is used for precise declaration purpose, while no compiler or symbolic execution engine is implemented for PRIML. We create PRIML because of the complex semantic model of C/C++ language. PRIML captures the primal semantic model of C/C++. We describe how PrivacyScope analyzes programs written in PRIML to reveal the core ideas. In addition to the PRIML examples in this section, we also show how PrivacyScope is implemented on top of Clang Static Analyzer and is applied to C/C++ enclave modules in section VI.

The Backus normal form grammar for PRIML is presented below. A PRIML program is composed of a sequence of statements. Statements consist of assignments and conditional branches. By design, all PRIML expressions are free from any side effects - they do not change the program state. We use "$\diamond_b$" and "$\diamond_u$" to represent binary (e.g. addition, subtraction and etc.) and unary operators (e.g. logical negations, XOR and etc), respectively. The statement get_secret($secret$) retrieves high variable from $secret$ while the statement declassify(exp) uncovers a value to the outside world (this is potentially observable for a malicious actor). For simplicity, we only consider expressions (constants, variables, etc.) which evaluate to 32-bit integer values. We also omit type-checking semantics of PRIML and assume all program under evaluation are well typed, e.g. binary operands are integers or variables.

$$
\begin{array}{lll}
stmt\ s & ::= & \text{skip} \mid var := exp \mid s_1; s_2 \\
 & & \mid \text{if } exp \text{ then } s_1 \text{ else } s_2 \\
exp\ e & ::= & exp \diamond_b exp \mid \diamond_u exp \mid var \\
 & & \mid \text{get\_secret}(secret) \mid v \mid \text{declassify}(exp) \\
\diamond_b & ::= & \text{typical binary operators} \\
\diamond_u & ::= & \text{typical unary operators} \\
value\ v & ::= & \text{32-bit unsigned integer}
\end{array}
$$

The operational semantics of a language specify unambiguously how the program should be interpreted in that language. We first define the base operational semantics before we specify program analysis. Each statement rule is of the form:

$$\frac{\text{computation}}{< \text{current state} >, \text{stmt} \rightsquigarrow < \text{end state} >, \text{stmt'}}$$

Rules are read from bottom up and left to right. Given a statement, PRIML interpreter pattern-matches at statement to find an applicable rule. For instance, the statement $x := e$ is interpreted according to *ASSIGN* rule. Then the interpreter evaluates the computation given on the top of the rule, and if successful, transitions to the end state. If no rule matches, then the machine halts abnormally. Δ maps a variable to its value for a given execution context, e.g. $\Delta[x]$ denotes the current value of variable x. We denote updating a context variable x with value v as $x \leftarrow v$. Thus, updating the value of variable x to the value 10 in context Δ is denoted as $\Delta[x \leftarrow 10]$. We denote evaluation of an expression e to a value v in the context of Δ by $\Delta \vdash e \Downarrow v$. PRIML interpreter evaluates expression e by matching e to an expression evaluation rule and performing the corresponding computation.

The complete operational semantics for PRIML are shown below. In addition, the context and statement $\Delta, skip$ indicates a termination.

$$\frac{v \text{ is input from } secret}{\Delta \vdash \text{get\_secret(}secret\text{)} \Downarrow v}\ \text{INPUT} \qquad \frac{}{\Delta \vdash var \Downarrow \Delta[var]}\ \text{VAR}$$

$$\frac{\Delta \vdash e \Downarrow v,\ v' = \diamond_u v}{\Delta \vdash \diamond_u e \Downarrow v'}\ \text{UNOP} \qquad \frac{}{\Delta \vdash v \Downarrow v}\ \text{CONST}$$

$$\frac{\Delta \vdash e_1 \Downarrow v_1,\ \Delta \vdash e_2 \Downarrow v_2,\ v' = v_1 \diamond_b v_2}{\Delta \vdash e_1 \diamond_b e_2 \Downarrow v'}\ \text{BINOP}$$

$$\frac{\Delta \vdash e \Downarrow v, \Delta' = \Delta[var \leftarrow v]}{\Delta, var := e \rightsquigarrow \Delta', skip}\ \text{ASSIGN}$$

$$\frac{\Delta \vdash e \Downarrow 1}{\Delta, \text{if } e \text{ then } s_1 \text{ else } s_2 \rightsquigarrow \Delta, s_1}\ \text{TCOND}$$

$$\frac{\Delta \vdash e \Downarrow 0}{\Delta, \text{if } e \text{ then } s_1 \text{ else } s_2 \rightsquigarrow \Delta, s_2}\ \text{FCOND}$$

$$\frac{\Delta' = \Delta, s_1}{\Delta, s_1; s_2 \rightsquigarrow \Delta', s_2}\ \text{COMP} \qquad \frac{}{\Delta, skip; s \rightsquigarrow \Delta, s}\ \text{SKIP}$$

$$\frac{\Delta \vdash e \Downarrow v, \text{declassify } v}{\Delta, declassify(e) \rightsquigarrow \Delta, skip}\ \text{DECLASS}$$

B. PrivacyScope Program Analysis

In this section, we describe how PrivacyScope analyzes programs written in PRIML language. We present how PrivacyScope works for PRIML to shed light upon how PrivacyScope works for C/C++. The objective of the analysis is to detect any violation of nonreversibility property in PRIML programs. PrivacyScope achieves this by combining taint tracking and forward symbolic execution. Taint tracking is used to track the flow of $high$ information from its sources to its sinks. Forward symbolic execution is used to reason about the behavior of the program under analysis given initial inputs. PrivacyScope represents the path condition of program execution as a logical formula, thus reducing the reasoning of a program's behavior to domain of logic. PrivacyScope represents variables symbolically and thus can examine program execution spanning multiple input space of the program at one time.

Component	Policy Check
$P_{get\_secret}(secret)$	t_n
$P_{const}()$	$\perp$
$P_{unop}(t), P_{assign}(t)$	t
$P_{binop}(t_1, t_2), P_{cond}(t_1, t_2)$	see Fig. 2
$P_{declassify\_check}(v, t, \pi, \tau_\Delta[\pi])$	see Alg. 1

TABLE I: PrivacyScope's policy for nonreversibility violation.

We express PrivacyScope in terms of the operational semantics of PRIML. To keep track of the taint status of each program value, we redefine values in PRIML to be tuples of the form $< v, \tau >$, where v is a value in the initial language, and τ is the taint status of v. τ is modeled by a security semi-lattice with join operation only shown in Fig. 1. In this semi-lattice, sensitive data is labeled by t1, t2 and more. If a variable is labeled by $\perp$, it means it is not sensitive. While if a variable is labeled by $\top$, it means it is tainted by two or more taint sources, so revealing it would not break nonreversibility. We also introduce a new mapping τ_Δ which maps variables to taint status.

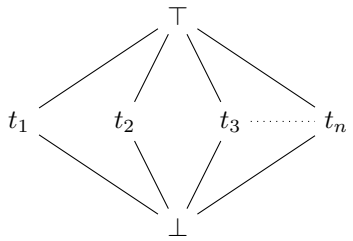

Fig. 1: Security semi-lattice for taint status

To enable forward symbolic execution in PRIML, we introduce the following changes to PRIML.

$value\ v$ ::= 32-bit unsigned integer | exp

π ::= Contains current constraints on symbolic variables due to path branches

These changes make partially evaluated symbolic expressions valid for a value in PRIML. Thus, when get_secret($secret$) is evaluated symbolically, it can return a symbol instead of a concrete value.

t_1	t_2	$P_{binop}(t_1, t_2), P_{cond}(t_1, t_2)$
$\top$	$\top$	$\top$
$\top$	t_2	$\top$
t_1	$\top$	$\top$
t_1	t_2	$\top\ if\ t_1\ \neq\ t_2\ else\ t_1$
t_1	$\perp$	t_1
$\perp$	t_2	t_2
$\perp$	$\perp$	$\perp$

Fig. 2: Truth table for $P_{binop}(t_1, t_2)$ and $P_{cond}(t_1, t_2)$

[1] $\neg$ operator negates the most recent added path constraint in π

Algorithm 1: $P_{declassify\_check}$

```
1: if t ≠ ⊥ or ⊤ then
2:     abort and report explicit leakage;
3: end if
4: if τ_Δ[π] ≠ ⊥ or ⊤ then
5:     if π is in hashmap hm then
6:         if v ≠ hm[π] then
7:             abort and report implicit leakage;
8:         else
9:             remove π in hm
10:            continue program analysis
11:        end if
12:    else
13:        hm[¬¹π] := v
14:        continue program analysis
15:    end if
16: else
17:    continue program analysis
18: end if
```

After introducing the aforementioned changes, Table I presents PrivacyScope policy for detecting nonreversibility violation. It introduces taint status into the system by marking up all values returned by get_secret($secret$) with different tainted status. Taint is then propagated through the program according to propagation rules. For constants, they are labeled as insensitive. For assignment and unary operations on a variable, they keep the same taint label for the variable. Fig. 2 shows taint label propagation rule for binary operation and conditional branches. The policy checks if $declassify(e)$ leaks secret whenever a value is revealed. Alg. 1 depicts $declassify(e)$ process. It first checks if the variable is labeled as sensitive. If yes, it reports an explicit leakage. If no, it then checks if the path constraint is sensitive. It uses a hashmap hm to assist with checking whether the revealed variable differs in distinct branches. Finally, at the end of the last path's interpretation, $declassify(e)$ checks if there is any item in hashmap hm. If so, it concludes that there is an implicit violation of nonreversibilitity. Note that, this last step is omitted in Alg. 1 to simplify the explanation. We summarize the PrivacyScope operational semantics as following.

$$\frac{v \text{ is a fresh symbol}}{\tau_\Delta, \Delta \vdash \text{get\_secret}(secret) \Downarrow < v, P_{secret}(secret) >}\ \text{PS-INPUT}$$

$$\frac{}{\tau_\Delta, \Delta \vdash var \Downarrow < \Delta[var], \tau_\Delta[var] >}\ \text{PS-VAR}$$

$$\frac{\tau_\Delta, \Delta \vdash e \Downarrow < v, t >,\ < v', t' > = \diamond_u < v, t >}{\tau_\Delta, \Delta \vdash \diamond_u e \Downarrow < v', P_{unop}(t) >}\ \text{PS-UNOP}$$

$$\frac{}{\tau_\Delta, \Delta \vdash v \Downarrow < v, P_{const}() >}\ \text{PS-CONST}$$

$$\frac{\tau_\Delta, \Delta \vdash e_1 \Downarrow < v_1, t_1 >,\ \tau_\Delta, \Delta \vdash e_2 \Downarrow < v_2, t_2 >}{\tau_\Delta, \Delta \vdash e_1 \diamond_b e_2 \Downarrow < v_1 \diamond_b v_2, P_{binop}(t_1, t_2) >}\ \text{PS-BINOP}$$

$$\frac{\tau_\Delta, \Delta \vdash e \Downarrow < v, t >, \Delta' = \Delta[var \leftarrow v],\ \tau'_\Delta = \tau_\Delta[var \leftarrow P_{assign}(t)]}{\tau_\Delta, \Delta, var := e \rightsquigarrow \tau'_\Delta, \Delta', skip}\ \text{PS-ASSIGN}$$

Statement	Δ	τ_Δ	abort
$h_1 := 2*$ get_secret($secret$)	$\{h_1 \to 2 * s_1\}$	$\{h_1 \to t_1\}$	$false$
$h_2 := 3*$ get_secret($secret$)	$\{h_1 \to 2 * s_1, h_2 \to 3 * s_2\}$	$\{h_1 \to t_1, h_2 \to t_2\}$	$false$
$x := h_1 + h_2$	$\{h_1 \to 2 * s_1, h_2 \to 3 * s_2, x \to 2 * s_1 + 3 * s_2\}$	$\{h_1 \to t_1, h_2 \to t_2, x \to \top\}$	$false$
$declassify(x)$	$\{h_1 \to 2 * s_1, h_2 \to 3 * s_2, x \to 2 * s_1 + 3 * s_2\}$	$\{h_1 \to t_1, h_2 \to t_2, x \to \top\}$	$false$
$declassify(h_1)$	$\{h_1 \to 2 * s_1, h_2 \to 3 * s_2, x \to 2 * s_1 + 3 * s_2\}$	$\{h_1 \to t_1, h_2 \to t_2, x \to \top\}$	$true$

TABLE II: Simulation of PrivacyScope detecting explicit leakage

Statement	Δ	π	τ_Δ	hm	abort
$h := 2*$ get_secret($secret$)	$\{h \to 2 * s\}$	$true$	$\{\pi \to \bot$ $h \to t_1\}$	$\{\varnothing\}$	$false$
if $h - 5 == 14$ then $declassify(0)$ else $declassify(1)$	$\{h \to 2 * s\}$	$[(2 * s) - 5 == 14]$	$\{\pi \to t_1$ $h \to t_1\}$	$\{\neg[(2 * s) - 5 == 14] \to 0\}$	$false$
if $h - 5 == 14$ then $declassify(0)$ else $declassify(1)$	$\{h \to 2 * s\}$	$\neg[(2 * s) - 5 == 14]$	$\{\pi \to t_1$ $h \to t_1\}$	$\{\neg[(2 * s) - 5 == 14] \to 0\}$	$true$

TABLE III: Simulation of PrivacyScope detecting implicit leakage

$$\frac{\tau_\Delta, \Delta \vdash e \Downarrow < e', t' >, \pi' = \pi \wedge (e' = 1), \quad \tau'_\Delta = \tau_\Delta[\pi \leftarrow P_{cond}(t', \tau_\Delta[t])]}{\pi, \tau_\Delta, \Delta, \text{if } e \text{ then } s_1 \text{ else } s_2 \rightsquigarrow \pi', \tau'_\Delta, \Delta, s_1} \text{ PS-TCOND}$$

$$\frac{\tau_\Delta, \Delta \vdash e \Downarrow < e', t' >, \pi' = \pi \wedge (e' = 0), \quad \tau'_\Delta = \tau_\Delta[\pi \leftarrow P_{cond}(t', \tau_\Delta[t])]}{\pi, \tau_\Delta, \Delta, \text{if } e \text{ then } s_1 \text{ else } s_2 \rightsquigarrow \pi', \tau'_\Delta, \Delta, s_2} \text{ PS-FCOND}$$

$$\frac{\tau'_\Delta, \Delta' = \tau_\Delta, \Delta, s_1}{\tau_\Delta, \Delta, s_1; s_2 \rightsquigarrow \tau'_\Delta, \Delta', s_2} \text{ PS-COMP}$$

$$\frac{}{\tau_\Delta, \Delta, skip; s \rightsquigarrow \tau_\Delta, \Delta, s} \text{ PS-SKIP}$$

$$\frac{\tau_\Delta, \Delta \vdash e \Downarrow < v, t >, P_{declassify\_check}(v, t, \pi, \tau_\Delta[\pi])}{\tau_\Delta, \Delta, declassify(e) \rightsquigarrow \tau_\Delta, \Delta, skip} \text{ PS-DECLASS}$$

Next, we present two simplified code segments written in PRIML to further explain the program analysis function of PrivacyScope.

Example 1. Consider the following program:

$$h_1 := 2 * \textit{get\_secret}(secret)$$
$$h_2 := 3 * \textit{get\_secret}(secret)$$
$$x := h_1 + h_2$$
$$declassify(x)$$
$$declassify(h_1)$$

We can easily see that declassifying x does not violate nonreversibility property. Knowledge of the value of x does not immediately reveal the value of the first $secret$. However, declassifying h_1 allows an attacker to infer the value of the first $secret$ by dividing the observed value with 2. Table. II presents a simulation of how PrivacyScope detects a leakage. Row 5 shows a leakage since the taint status of h_1 is t_1, while row 4 does not because the taint status of x is $\top$.

Example 2. Consider the following code snippet:

$$h := 2 * \textit{get\_secret}(secret)$$
$$\text{if } h - 5 == 14 \text{ then } declassify(0) \text{ else } declassify(1)$$

By observing the declassified output, an attacker can easily infer if h is equal to 19 or not and, ultimately recover the $secret$. Table. III illustrates how PrivacyScope detects implicit leakage. Row 3, for instance, reports a leakage since the taint status for π is t_1, and the value retrieved from the hashmap hm is 0 which is different from what $declassify$ is outputting (1). Row 2 of Table. III, on the other hand, does not report a leakage even though the taint status for π is t_1. Because nothing is stored in the hashmap hm before the interpretation.

C. Incorporating PrivacyScope in an Intel SGX enclave

Intel SGX enclave modules are typically written in C/C++. Therefore, the following part incorporates aforementioned core ideas written in PRIML and presents how PrivacyScope is integrated into Intel SGX ecosystem. Each Intel SGX enclave declares one or more entry points into the enclave. Referred to as ECALLS by the Intel SGX SDK, these interfaces allow untrusted outside applications access trusted code running inside the enclave. An enclave may also have OCALLS, which allow trusted enclave code to call out to the untrusted application. To configure an application to run in an Intel SGX enclave, an enclave interface definition file is created. An EDL file resembles a traditional C header file and contains prototype declarations for all ECALL and OCALL interfaces. The enclave EDL file defines how data is marshalled between enclaves trusted code and the outside untrusted applications.

To pass secret data to enclave code for further processing, enclave uses ECALL type interface. Similar to C function prototypes, the ECALL interface parameters are annotated with attributes like $[in]$ and/or $[out]$. A parameter with $[in]$ attribute is used for marshalling data from outside untrusted application into the enclave. So in our example, secret data is passed into enclave via $[in]$ parameter(s). Interface parameters with $[out]$ attribute are used to marshal data from inside the enclave to the outside untrusted application. The Intel SGX SDK abstracts out the details of the data marshalling by

generating the necessary proxy code. In short, $[in]$ parameters correspond to get_secret($secret$) in the previous section.

Prior to performing code analysis, PrivacyScope processes an XML configuration file, provided by user, containing function names that the user is interested in evaluating. PrivacyScope also extracts information included in the SGX EDL configuration file. Following a quick initialization step, PrivacyScope analyzes program code using the approach outlined in the previous section and generates a report summarizing the outcome of the code analysis including any violations of non-reversibility property. For *explicit* information leakage cases, the report describes how program output can be used to infer its (secret) input thus assisting developers in securing their code. For implicit information leakage, the report provides path conditions and returns results which can result in leakage of secret data.

VI. Evaluations

A. Implementation

For our proof-of-concept prototype, we use the Intel SGX SDK version 2.0 to construct a trusted application running in a SGX enclave powered by an Intel NUC, running under Ubuntu 14.04 TLS. Built on top of Clang v7.0.0, we build a prototype of PrivacyScope by adding over 1 KLOC. We have evaluated the prototype of PrivacyScope by porting open source machine learning programs written in C/C++ to Intel SGX enclaves and analyzing the enclave modules for data leakage.

B. Illustration: a Leakage Example in C

To illustrate PrivacyScope operation on real Intel SGX enclave module written in C, we provide an illustrative example here. For simplicity, our example neglects decryption of secret data. However, PrivacyScope does consider decryption of encrypted secret data, it records decryption function names from Intel SGX IPP library in a predefined list. And when PrivacyScope meets predefined decryption functions, it assigns the symbolic value of secret data to decrypted secret data. For illustration, we define ECALL function in EDL file for processing secret data as $int\ enclave\_process\_secret(\ [in]\ secrets,\ [out]\ output)$. Source code of $enclave\_process\_secret$ function is shown in Listing 1. In this simplified example, we can easily see that $secrets[0]$ is explicitly leaked and $secrets[1]$ is implicitly leaked.

```
1 int enclave_process_data(char *secrets, char *output
      ){
2     int temporary = secrets[0] + 100;
3     output[0] = temporary + 1;
4     if(secrets[1] == 0)
5         return 0;
6     else
7         return 1;
8 }
```

Listing 1: Code snippet of illustrative example in C

Next we show how PrivacyScope explores the illustrative example and identifies explicit and implicit privacy leakage using the symbolic execution engine of Clang Static Analyzer. First, we need to define the state that the symbolic execution engine must maintain. We define the state as a 4-tuple $(stmt, env, \sigma, \pi)$ similar to [15] where:

- $stmt$ represents the next statement in source code to be evaluated. In our illustrative example, a $stmt$ can be an assignment, a conditional branch, or a return statement.
- env is the environment which maps from $lvalue$ (an $lvalue$ is an expression with an object type according to C standard [19]) expressions to memory regions (abstract representation of memory objects) reg_i. A memory region can be the subregion of another region. And when a variable of array type is defined, it has subobjects called elements. For array elements, we have $ElementRegion$ like $reg_i[0]$ with its super region reg_i to represent the array region. As for when a pointer is pointing to some unknown memory block, we have $SymRegion$ for representing the memory block pointed to by the symbolic pointer. $SymRegion$ represents a region that serves as an alias for either a real region, a NULL pointer, etc. It essentially is used to map the concept of symbolic values into the domain of regions.
- σ is a store which maps from memory regions to concrete values, symbolic values α_i or memory regions reg_i.
- π denotes path constraints on symbolic values. At the beginning of a symbolic execution, π is set to True. As the symbolic execution engine explores the program statements, π grows when branches are met and assumptions on taking any branch are recorded in π as a formula. This formula indicates how execution can reach any $stmt$ in the program code under analysis.

Depending on $stmt$, the symbolic engine of Clang Static Analyzer changes states as following:

- The evaluation of an assignment $x = e$ updates the environment env and store σ. e can be any legal expression involving unary or binary operators over symbolic or concrete values. If e contains unknown $lvalue$ expressions in the context of current execution state, new reg_is are initialized for the unknown expressions in updated env and mapping between newly initialized reg_i and context of current execution state is created in updated σ. Assuming e_s is the symbolic expression of evaluating e, e_s is associated with x in updated env.
- The evaluation of a conditional branch $if\ e\ then\ stmts_{true}\ else\ stmts_{false}$ affects the path constraints π. When a conditional branch is met, the symbolic engine will fork and create two new execution states, one with path condition $\pi_{true} = \pi \wedge e_s$ and the other with $\pi_{false} = \pi \wedge \neg e_s$, where e_s is the symbolic expression by evaluating e. Symbolic engine will follow the two newly created execution states one by one.

Execution State	Line #	$stmt$	env	σ	π
A	2	int temporary = secrets[0] + 100;	$\emptyset$	$\emptyset$	True
B	3	output[0] = temporary + 1;	$secrets \rightarrow reg_0$ $secrets[0] \rightarrow reg_1$ $temporary \rightarrow reg_1 + 100$	$reg_0 \rightarrow SymRegion$ $reg_1 \rightarrow reg_0[0]$	True
C	4	if (secrets[1] == 0)	$secrets \rightarrow reg_0$ $secrets[0] \rightarrow reg_1$ $temporary \rightarrow reg_1 + 100$ $output \rightarrow reg_2$ $output[0] \rightarrow reg_1 + 101$	$reg_0 \rightarrow SymRegion$ $reg_1 \rightarrow reg_0[0]$ $reg_2 \rightarrow SymRegion$	True
D	5	return 0;	$secrets \rightarrow reg_0$ $secrets[0] \rightarrow reg_1$ $temporary \rightarrow reg_1 + 100$ $output \rightarrow reg_2$ $output[0] \rightarrow reg_1 + 101$ $secrets[1] \rightarrow reg_3$	$reg_0 \rightarrow SymRegion$ $reg_1 \rightarrow reg_0[0]$ $reg_2 \rightarrow SymRegion$ $reg_3 \rightarrow reg_0[1]$	$reg_0[1] == 0$
E	7	return 1;	$secrets \rightarrow reg_0$ $secrets[0] \rightarrow reg_1$ $temporary \rightarrow reg_1 + 100$ $output \rightarrow reg_2$ $output[0] \rightarrow reg_1 + 101$ $secrets[1] \rightarrow reg_3$	$reg_0 \rightarrow SymRegion$ $reg_1 \rightarrow reg_0[0]$ $reg_2 \rightarrow SymRegion$ $reg_3 \rightarrow reg_0[1]$	$reg_0[1] \neq 0$

TABLE IV: Exploration of illustrative example

Box 1: Report generated by PrivacyScope for the illustrative example.

illustrative_example.c:5:9: warning: The memory region 'reg_$3 <char element {SymRegion {reg_$0 <char* secrets>}, 1 S64b, char}>' and its concrete value '{0, 0}' breaks privacy and implicitly leaks sensitive data!
illustrative_example.c:8:6: warning: The memory region 'SymRegion{reg_$1 <char* output>}' and its symbolic value '(reg_$2 <char* secrets>}, 0 S64b, char}>) + 101' breaks privacy and explicitly leaks sensitive data!

Table IV presents the symbolic exploration of our illustrative example. Initially (execution state A), the path condition is set to true and env and σ are null. During the evaluation of line 2 right-hand side (RHS), the first evaluated expression is $secrets$, so a new region reg_0 is generated and associated with $secrets$ in env and reg_0 is mapped to $SymRegion$ in σ. Then it follows the evaluation of $secrets[0]$ which leads to a new expression region map of $secrets[0]$ to reg_1 in env. σ is also updated with a new map of reg_1 to $reg_0[0]$. After the evaluation of RHS, the result, $reg_1 + 100$, is associated with $temporary$. Next (execution state B), the evaluation of RHS of line 3 returns $reg_1 + 101$ within the execution context and the result is associated with the evaluation of line 3's left-hand side (LHS). The evaluation of line 3's LHS brings a new region reg_2 and it is associated with $output$ in env. Meanwhile, reg_2 is mapped to $SymRegion$ in σ. Besides, the result of line 3's RHS, $reg_1 + 101$, is associated with $output[0]$ in env. When a conditional branch is met (execution state C), the engine will fork into two execution states (D and E) with opposite path constraints on the comparison statement. During execution of state C, a new region reg_3 is assigned to $secrets[1]$ in env and reg_3 is an $ElementRegion$ of reg_0. By evaluating the comparison statement, two opposite path conditions, $reg_0[1] == 0$ and $reg_0[1] \neg = 0$, are added into π of execution states D and E. The evaluation of return statement calls the procedure of policy check similar to previous $P_{declassify\_check}$.

When PrivacyScope starts exploring the target function *enclave_process_data*, it first goes into EDL file and fetches parameters as specified by user predefined rules. If no rules are predefined, the default action is to mark $[out]$ attribute parameters as potential leaking point, and $[in]$ attribute parameters as secrets. Then PrivacyScope starts exploration of source code as described in previous paragraphs. During the exploration, PrivacyScope introduces taint status to secret variables and propagates the tainting as mentioned in program analysis for PriML. When $enclave\_process\_data$ function returns or ends, PrivacyScope performs a policy check similar to $P_{declassify\_check}$. For explicit privacy leakage check, PrivacyScope checks $[out]$ parameters to see their taint status. In the illustrative example case, $output[0]$ is tainted by t_1, so $output[0]$ explicitly leaks the value of $secrets[0]$. For implicit privacy leakage check, PrivacyScope utilizes hashmap hm and find that the returned values are different for different π which branch on subobject of $secrets$. The warning report generated for the illustrative example is shown in Box 1.

C. Performance Evaluation

Our primary objective is to detect any violation of non-reversibility property in a trusted code executing inside a SGX enclave. However, current version of SGX SDK does not support easy migration of legacy application even if it

Open Source ML Code	Size (LoCs)	Execution Time (sec.)
LinearRegression	161	2.549s
Kmeans	179	4.654s
Recommender	117	1.758s

TABLE V: Performance evaluation

is written in C/C++ and there is no existing open source implementation of ML algorithms implemented inside enclave using the Intel SGX SDK. Thus, we had to port open source ML algorithms so that they can be executed inside a SGX enclave. We selected three popular open source ML projects from the public Github repository and ported them using Intel SGX SDK. They included LinearRegresssion, Kmeans and Recommender [27]–[29]. To evaluate the efficacy of PrivacyScope, we inserted malicious code inside the ported ML code. The results were examined by the authors and the efficacy of PrivacyScope solution was verified by the authors manually. During this process, we also detected multiple preexisting secret leakage in the open source Recommender implementation. We provide a detailed explanation of these findings in the next section. Table V summarizes performance data, including the PrivacyScope code analysis time for each of the three open source ML projects. The analysis time was measured using the Linux OS built-in time utility. The total execution time was computed by summing up the usr and sys times.

D. Case Studies

The goal of PrivacyScope is to assist users and developers in identifying potential information leakage vulnerabilities in program code intended to run in a TEE-protected environment. PrivacyScope accomplishes this objective by identifying any violations of the nonreeversibility property. In this section, we present two case studies to demonstrate PrivacyScope's capabilities by analyzing the behavior of two open source ML programs. These two examples by no means cover all the possible scenarios. In the first case, PrivacyScope uncovers implementation defects caused by inadvertent coding errors during software development. In the second case, we illustrate how PrivacyScope can detect malicious code inserted into the codebase by a malicious actor.

1) Finding information leakage in Recommender: Our first case study is the analysis of a C library for product recommendations/suggestions using collaborative filtering (CF) [27]. Recommender analyzes and learns from collective feedback of a large number of users. It then uses user preference to predict and recommend the most appropiate products for a particular user. We found 6 violations of the nonreversibility property in this open source ML project. We detail this process in Appendix [30].

2) Verifying effectiveness of PrivacyScope in Kmeans: Our second case study is mimicking a malicious enclave writer and embedding sensitive data leakage logic inside enclave programs. We add explicit and implicit leakage logic to open source machine learning program Kmeans [29]. We show the details of how we insert malicious logics in Appendix [30].

VII. Related Work

A. Information Flow Analysis Methods

Use of information flow analysis to detect information leakage within programs has been the subject of much research in the past decades. Language-specific methods typically augment type systems so that they can statically check the flow of private data within programs that manipulate the data. [9], for example, integrates information flow analysis into the Java language type system, while [9], [25], [26] propose innovative analysis methods for imperative and concurrent language. [21] detects security vulnerabilities in hardware design written in HDL. In these solutions, they focus on addressing the noninterference property in programs.

In addition to aforementioned information flow analysis, a plethora of methods have focused on detecting, measuring and understanding the nature of privacy leakage on different platforms. Static and dynamic tainting analysis are two major categories of methods for detecting privacy leakage. Dynamic methods typically monitors the system at runtime and examine the system as it executes the code. Static methods on the other hand use compile time analysis to predict the impact of code execution on private data. Static methods can have a higher false positive rate as compared to dynamic methods [23]. [20] leverages dynamic tainting to detect intentional leakage of private data in the Android operating system environment while [22] examines system-wide information flow for malware detection in the Windows platform. Finally, [8] uses model checking to verify the information flow properties of code running in a trusted enclave. Table VI summarizes key recent research work in the area of information analysis methods, highlighting the target environment for each approach.

B. Secure Systems on Trusted Hardware

There have been many recent advances in the area of trusted hardware platforms including the development of commercial off-the-shelf secure processors. ARM TrustZone [31] offers a processor capable of executing in a secure world as well as a normal world with isolated address spaces. TPM+TXT [32] provides attestation on the execution state of a platform, but all privileged software must execute in the trusted computing base. SGX [33]–[36], an extension to the Intel Architecture, offers confidentiality and integrity guarantees via a trusted processor without requiring any trust on the part of infrastructure software.

Multiple secure systems have been recently built on top of these trusted hardware platforms [2]–[4], [37], [38]. VC3 [2] and Opaque [4] offer SGX-protected data processing platform, assuming that, the code executing inside each enclave is trusted. Thus, their confidentiality guarantee is based on the assumption that enclave code does not leak secrets. In these cases, PrivacyScope can be used to confirm this assumption. Ryoan [3] and Chiron [38] utilize sandboxing to prevent untrusted enclave module from leaking secret data from side channels. To be adopted, these methods require a certain level of trust to be established between users, service providers

	Approach			Target Leakage							Target System
	Type System	Dynamic Analysis	Static Analysis	Explicit flow		Implicit flow					
				NonInt*	NonRev*	NonInt*	NonRev*	Termination	Timing	Probability	
Taintdroid [20]		✓		✓							Android
HDL Checker [21]	✓			✓		✓					Hardware Design
Panorama [22]		✓		✓							Windows OS
Androidleaks [23]			✓	✓							Android
Covert Flow Checker [24]	✓							✓			An Imperative Language
Timing Leaks Transformer [25]	✓								✓		An Imperative Language
Multithread Possibilistically NonInt*[26]	✓									✓	A Concurrent Language
Jflow [9]	✓			✓		✓					Java
Moat [8]	✓			✓		✓					Intel SGX Enclave
This work			✓		✓		✓				Intel SGX Enclave

* NonInt is short for noninterference, NonRev is short for nonreversibility.

TABLE VI: Systematic approaches for detecting secret leakage

and their solution. PrivacyScope can strengthen users' trust in these scenarios. All of these methods require a modicum of trust between the user and the trusted computing platform. PrivacyScope is designed to address this concern in the Intel SGX architecture.

C. Privacy Leakage in Machine Learning

Using ML to train models on big data poses many privacy challenges. ML models can uncover and expose surprising and unexpected personally identifiable information such as relationships and associations violating privacy. [39]–[41] take a first step to conceptualize privacy in the new era and enforce data use rules through designing new policy specification languages and corresponding enforcing systems. [40] creates a language for specification of origin-based privacy rules and implements a prototype of a type system to enforce such policies. In [41], the authors present LEGALEASE - a language to specify privacy specifications and restrict how private data must be handled. [39] presents Thoth which provides data use policies enforcement through a kernel-level compliance layer. PrivacyScope can integrate such policies and enforce more rules other than nonreversibility in the future.

VIII. Discussion and Future Work

A. Covert and Side Channels

In this section, we briefly highlight potential covert channels that can be exploited to leak private data in the Intel SGX computing environment. Each covert channel compromises PrivacyScope's security goals. We will address mitigation techniques to safeguard against these attacks in future works.

- *Timing channel*: Malicious actor can infer secret through recording the time spent for program executions. PrivacyScope can be extended to simulate the execution time for program paths and detect if execution time depends on secret in the future.
- *Probabilistic channel*: Malicious actor can infer secret through observing probability distribution of declassified data.
- *Power channel*: Malicious actor can measure the power consumption cost for program executions and infer secret.

PrivacyScope does not address side channels brought by hardware limitations on Intel SGX processor itself. We consider these limitations orthogonal to PrivacyScope and we list them as following.

- *SGX page faults*: Privileged software, e.g. OS or hypervisor, can maliciously control page tables of an enclave to observe a memory access pattern of enclave program execution.
- *Cache timing*: Side channel exists for two processes running on the same core. These two processes can use cache timing to infer knowledge of the other.
- *Address bus monitoring*: While all processed data is encrypted prior to exiting the SGX processor package, a malicious user using sniffer or a modified RAM chip can monitor the address bus and achieve a memory access pattern side channel.

B. Prior Knowledge on User Data

When the adversary has prior knowledge of user private data (e.g. knows the distribution of variable values), PrivacyScope requires that knowledge to be incorporated in the model specification to ensure the soundness of the privacy leakage analysis. For instance, given the function $F(A, B) = A + B$, where A is a privacy-sensitive scalar variable and B has a value of zero 99% of the time, and 1 otherwise. Then the attacker can conclude with a high degree of confidence (i.e. 99%) that the output value is the same as the value of the privacy-sensitive input variable A. To mitigate this problem, the users of PrivacyScope are required to incorporate that knowledge by extending PRIML language and the analysis engine so that it can handle with new semantics.

C. Limitations and Future Work

PrivacyScope is built on top of symbolic execution engine and symbolic execution is known to have limitation on scalability. Although enclave code does not have large size, they will become larger in the future. Also, our design needs enclave writer and cloud service provider to send user the source code of enclave for validation. This may hurt the intellectual property of cloud service provider. In the future, we plan to analyze on binary enclave code directly instead of C/C++ source code. This would protect the intellectual property of cloud service provider.

IX. Conclusion

In this work, we formally define new nonreversible property on top of classical noninterference property, which is more suitable for machine learning programs. Using our newly proposed language, PRIML, we describe our innovative program analysis approach using formal semantics. PRIML's formal semantics can be extended by users who wish to introduce their own specialized notion of nonreversibility. We design and implement PrivacyScope, a prototype static analysis tool that implements the rules as defined in PRIML to detect nonreversibility violation in programs executing inside Intel SGX enclave. We show the efficacy of our prototype by applying PrivacyScope to find sensitive data leakage and maliciously embedded code in open source machine learning programs.

Acknowledgment

This work was supported in part by the Office of Naval Research under grant N00014-19-1-2621, US National Science Foundation under grant CNS-1837519, and Virginia Commonwealth Cyber Initiative.

References

[1] C. Gentry and D. Boneh, *A fully homomorphic encryption scheme*, vol. 20. Stanford University Stanford, 2009.

[2] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 38–54, IEEE, 2015.

[3] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data.," in *OSDI*, 2016.

[4] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform.," in *NSDI*, pp. 283–298, 2017.

[5] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Security and Privacy (SP), 2015 IEEE Symposium on*, pp. 640–656, IEEE, 2015.

[6] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 2421–2434, ACM, 2017.

[7] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside sgx enclaves with branch shadowing," in *26th USENIX Security Symposium, USENIX Security*, pp. 16–18, 2017.

[8] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave programs," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pp. 1169–1184, ACM, 2015.

[9] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 228–241, ACM, 1999.

[10] F. Liu, H. Wu, and R. B. Lee, "Can randomized mapping secure instruction caches from side-channel attacks?," in *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, p. 4, ACM, 2015.

[11] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[12] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select—a formal system for testing and debugging programs by symbolic execution," *ACM SigPlan Notices*, vol. 10, no. 6, pp. 234–245, 1975.

[13] J. B. Kam and J. D. Ullman, "Monotone data flow analysis frameworks," *Acta Informatica*, vol. 7, no. 3, pp. 305–317, 1977.

[14] F. E. Allen and J. Cocke, "A program data flow analysis procedure," *Communications of the ACM*, vol. 19, no. 3, p. 137, 1976.

[15] Z. Xu, T. Kremenek, and J. Zhang, "A memory model for static analysis of c programs," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, Springer, 2010.

[16] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Security and Privacy, 1982 IEEE Symposium on*, IEEE, 1982.

[17] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[18] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and privacy (SP), 2010 IEEE symposium on*, pp. 317–331, IEEE, 2010.

[19] "Understanding lvalues and rvalues in c/c++." https://eli.thegreenplace.net/2011/12/15/understanding-lvalues-and-rvalues-in-c-and-c.

[20] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Transactions on Computer Systems (TOCS)*, 2014.

[21] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh, "Verification of a practical hardware security architecture through static information flow analysis," in *ACM SIGARCH Computer Architecture News*, vol. 45, pp. 555–568, ACM, 2017.

[22] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 116–127, ACM, 2007.

[23] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale," in *International Conference on Trust and Trustworthy Computing*, pp. 291–307, Springer, 2012.

[24] D. Volpano and G. Smith, "Eliminating covert flows with minimum typings," in *Computer Security Foundations Workshop, 1997. Proceedings., 10th*, pp. 156–168, IEEE, 1997.

[25] J. Agat, "Transforming out timing leaks," in *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 40–53, ACM, 2000.

[26] D. Volpano and G. Smith, "Probabilistic noninterference in a concurrent language 1," *Journal of Computer Security*, 1999.

[27] https://github.com/GHamrouni/Recommender/tree/master/src.

[28] https://github.com/aluxian/CPP-ML-LinearRegression.

[29] https://github.com/pramsey/kmeans.

[30] https://github.com/Ruide/ICDCS_Appendix/blob/master/ICDCS__PrivacyScope_Appendix.pdf.

[31] T. Alves and D. Felton, "Trustzone: Integrated hardware and software security-enabling trusted computing in embedded systems (july 2004)."

[32] D. Grawrock, *Dynamics of a Trusted Platform: A building block approach*. Intel Press, 2009.

[33] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for cpu based attestation and sealing," in *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, vol. 13, ACM New York, NY, USA, 2013.

[34] V. Costan and S. Devadas, "Intel sgx explained.," *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016.

[35] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions.," *HASP@ ISCA*, vol. 11, 2013.

[36] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution.," *HASP@ ISCA*, vol. 10, 2013.

[37] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "Case: Cache-assisted secure execution on arm processors," in *Security and Privacy (SP), 2016 IEEE Symposium on*, pp. 72–90, IEEE, 2016.

[38] T. Hunt, C. Song, R. Shokri, V. Shmatikov, and E. Witchel, "Chiron: Privacy-preserving machine learning as a service," *arXiv preprint arXiv:1803.05961*, 2018.

[39] E. Elnikety, A. Mehta, A. Vahldiek-Oberwagner, D. Garg, and P. Druschel, "Thoth: Comprehensive policy compliance in data retrieval systems.," in *USENIX Security Symposium*, pp. 637–654, 2016.

[40] H. Nissenbaum, S. Benthall, A. Datta, M. C. Tschantz, and P. Mardziel, "Origin privacy: Protecting privacy in the big-data era," tech. rep., NEW YORK UNIVERSITY New York United States, 2018.

[41] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Tsai, and J. M. Wing, "Bootstrapping privacy compliance in big data systems," in *Security and Privacy (SP), 2014 IEEE Symposium on*, pp. 327–342, IEEE, 2014.