2020 IEEE 33rd Computer Security Foundations Symposium (CSF)

PASAPTO: Policy-aware Security and Performance Trade-off Analysis – Computation on Encrypted Data with Restricted Leakage

Andreas Fischer*, Jonas Janneck*, Jörn Kussmaul*, Nikolas Krätzschmar*, Florian Kerschbaum†, Eric Bodden‡
* SAP Security Research, pasapto@andreasfischer.net
† School of Computer Science, University of Waterloo, Canada, fkerschbaum@uwaterloo.ca
‡ Department of Computer Science, Paderborn University, Heinz Nixdorf Institute & Fraunhofer IEM, eric.bodden@upb.de

***Abstract*—This work considers the trade-off between security and performance when revealing partial information about encrypted data computed on. The focus of our work is on information revealed through control flow side-channels when executing programs on encrypted data. We use quantitative information flow to measure security, running time to measure performance and program transformation techniques to alter the trade-off between the two. Combined with information flow policies, we perform a policy-aware security and performance trade-off (PASAPTO) analysis. We formalize the problem of PASAPTO analysis as an optimization problem, prove the NP-hardness of the corresponding decision problem and present two algorithms solving it heuristically.**

We implemented our algorithms and combined them with the Dataflow Authentication (DFAuth) approach for outsourcing sensitive computations. Our DFAuth Trade-off Analyzer (`DFATA`) takes Java Bytecode operating on plaintext data and an associated information flow policy as input. It outputs semantically equivalent program variants operating on encrypted data which are policy-compliant and approximately Pareto-optimal with respect to leakage and performance. We evaluated `DFATA` in a commercial cloud environment using Java programs, e.g., a decision tree program performing machine learning on medical data. The decision tree variant with the worst performance is 357% slower than the fastest variant. Leakage varies between 0% and 17% of the input.

I. Introduction

Cloud services provide on-demand access to cost-efficient computer resources such as data storage and computing power. However, when using these services, data is at risk of being stolen by attackers observing the cloud, e.g., malicious administrators. To ensure the confidentiality of sensitive data, encryption can be applied to the data prior to transferring it to the cloud. But, in order to use the cloud's computing power without compromissing data confidentiality, the cloud must operate on encrypted data.

Unfortunately, efficient computation on encrypted data without side-channels remains an open problem. On the one hand, cryptographic techniques such as fully homomorphic encryption (FHE) [22] enable arbitrary computations that do not reveal any information about the data computed on, but suffer high computational costs [23]. On the other hand, Trusted Execution Environments (TEEs) such as Intel Software Guard Extensions (SGX) entail only little computational overhead, but are vulnerable to side-channel attacks. For example, it has been demonstrated that the control flow executed inside SGX enclaves can be inferred from untrusted programs [35]. In fact, SGX "is not designed to handle side-channel attacks" [16] and it is up to the developers to build their enclaves accordingly.

For some applications meaningful security can only be achieved when all side-channels are eliminated. Consider for example cryptographic primitives such as square-and-multiply algorithms used for modular exponentiation in public key cryptography. If private key bits are leaked through side-channels [12], all security relying on the secrecy of the private key is lost.

A broad class of side-channels can be avoided by producing constant-time code not making any memory accesses or control flow decisions based on secret data. Some cryptographic primitives have even been designed such that an implemenation likely possesses these properties [8]. However, for more complex applications these properties cannot be achieved without causing prohibitive performance. Consider for example Dantzig's simplex algorithm [17] for solving linear optimization problems. This algorithm terminates as soon as objective values of the current solution can no longer be improved. It is extraordinarily efficient in practice but its worst-case running time is exponential in the problem size [30]. If we are to eliminate all side-channels, we must also prevent the termination condition from leaking. Thus, any invocation must have exponential running time. Since this behaviour is impractical for any non-trivial input, engaging in a trade-off between security and performance seems justified.

This trade-off does not only occur when outsourcing sensitive computations, but the same trade-off can be made in many other types of computation on encrypted data. For example it also applies to secure multi-party computation (MPC) protocols, which allow parties to jointly compute a function while keeping their inputs private. By declassifying (i.e., making public) partial information such as intermediate results, expensive MPC computations can be avoided and the performance of MPC protocols can be improved.

In this work, we consider the trade-off between security and performance of control flow side-channels when executing programs on encrypted data. Our motivation for focussing on control flow is based on the expectation that the disclosure of control flow information, e.g., rather than computing a circuit

© 2020, Andreas Fischer. Under license to IEEE.
DOI 10.1109/CSF49147.2020.00024

```
1  int p1(int x)
2    if (e)
3      y = f(x)
4    else
5      y = g(x)
6    return y
```

```
1  int p2(int x)
2    y1 = f(x)
3    y2 = g(x)
4    # obliviously select
5    # y1 or y2 based on e
6    y = select(e, y1, y2)
7    return y
```

Figure 1: Two semantically equivalent variants of a program.

as in FHE, results in significant performance gains.

Consider for example the two variants of a program provided in Figure 1. The first program p1 reveals the control flow – more precisely the boolean result of the conditional expression e – to an attacker capable of observing the executed control flow. Based on the result of e, only either f or g is computed. The second program p2 computes both f and g and combines the result by invoking an oblivious select function. Because no control flow decisions can be observed, the attacker does not learn the result of e. Since p2 has to perform $c(f) + c(g)$ computations while p1 has to perform only $\max\{c(f), c(g)\}$ computations, we expect p2's performance to be inferior to that of p1.

More generally, a trade-off for a program is determined by selecting for each control flow decision whether it may be *revealed* or must be *hidden*. Since the number of program variants is exponential in the number of control flow decisions and not all selections have the same impact on security and performance, it is impracticable for developers to make such selections manually for any non-trivial program. Our analysis uses program transformation techniques to hide control flow decisions and explore the security-performance trade-off.

Once a specific trade-off has been chosen, formal verification techniques [2], [38], [44] can be used to ensure that compilers do not introduce additional side-channels. However, these techniques are orthogonal to our work since they verify the compliance of an implementation to a given specification (i.e., ideal functionality in MPC) whereas we modify the specification by altering the security-performance trade-off.

To enable strong security guarantees, our analysis incorporates information flow policies, which allow developers to define varying sensitivity levels on data. By defining the appropriate sensitivity level, developers can also completely prevent data from being revealed during program execution.

We formalize the problem of policy-aware security and performance trade-off (PASAPTO) analysis as an optimization problem and prove the NP-hardness of the corresponding decision problem. To make the problem tractable we develop two heuristic algorithms computing an approximation of the Pareto front of the optimization problem.

In order to investigate the efficacy of our heuristics, we implemented and empirically evaluated them using Dataflow Authentication (DFAuth) [20]. DFAuth computes on encrypted data in the cloud using a combination of homomorphic encryption and a small Trusted Module, for example an SGX enclave. Compared to an SGX-only solution, DFAuth minimizes the exposure to software vulnerabilities by providing a small and program-independent trusted code base. Our DFAuth Trade-off Analyzer (DFATA) takes as input a program operating on plaintext data and an associated information flow policy. It outputs semantically equivalent program variants operating on encrypted data which are policy-compliant and approximately Pareto-optimal with respect to security and performance.

We evaluated DFATA in a commercial cloud environment on two use cases: an electronic sealed-bid auction program and a decision tree program performing machine learning on sensitive medical data. We chose these programs such that we are able to explore all program variants in order to evaluate the effectiveness of our heuristics, but complex enough to be non-trivial in the context of computation on encrypted data (e.g., FHE, MPC). Security was measured using quantitative information flow (QIF), performance was measured using wall-clock running time. The results of our experiments confirm that our heuristics indeed adequately explore the search space of security-performance trade-offs.

In the first experiment, the leakage of program variants varies between 0% and 2% of the program input. The variant with the worst performance is 99% slower than the fastest variant. Our first (second) heuristic inspected 9% (7%) of all variants and output 18 (13) variants in the solution set. The heuristic solution is 5% (20%) worse than the ideal solution obtained using exhaustive search. In the second experiment, the leakage varies between 0% and 17% of the input. The variant with the worst performance is 357% slower than the fastest variant. Our first (second) heuristic inspected 1% (5%) of all variants and output 14 (11) in the solution set. The heuristic solution is 2% (14%) worse than the ideal solution.

In summary, our contributions are:

- We formalize the problem of policy-aware security and performance trade-off (PASAPTO) analysis as an optimization problem and prove the NP-hardness of the corresponding decision problem.
- We present two heuristic algorithms computing an approximation of the Pareto front of the optimization problem: a greedy heuristic providing fast convergence and a genetic algorithm providing well distributed trade-offs.
- We adjust an existing QIF analysis to capture the adversarial information flow for each variable of a program such that we can support variable-based information flow policies.
- We implemented our algorithms and evaluated them on Java programs in a commercially available cloud.

In the next section, we introduce various definitions. Section III explains how we quantify adversarial information flow resulting from control flow observations. Section IV shows how to quantify variable-specific information flow and establish policy-compliance. Section V presents the PASAPTO optimization problem and our two heuristics. In section VI we apply our PASAPTO analysis to DFAuth. Section VII provides details about the implementation of DFATA. Section VIII presents the results of our experiments. Section IX discusses related work before Section X concludes our work.

II. Definitions

A. Programs and Computation

We model a program as a transition system $P = (S, Tr, I, F)$ consisting of a set of *possible program states* S, a set of program *transitions* Tr, a set of *initial states* $I \subseteq S$ and a set of *final states* $F \subseteq S$. We refer to the *set of all programs* as $\mathcal{P}$.

A program state $s \in S$ contains an *instruction pointer* and a map assigning each program variable a value from its domain. We refer to the *set of variables* of a program as V. Without loss of generality, we assume all variables to be integers in the range $D_n := [-2^{n-1}, 2^{n-1} - 1] \subset \mathbb{Z}$.

Each transition corresponds to a program *statement* as written in a programming language. We consider a deterministic, imperative programming language with assignments, conditional expressions, et cetera. We refer to a transition corresponding to a program statement containing a control flow decision as a *control flow transition*. By $T \subseteq Tr$ we denote the set of all control flow transitions.

B. Adversary Model

We consider a passive adversary who continuously observes the instruction pointer of the program state $s \in S$ during an execution of a program. We assume the adversary knows the program text, i.e., any transition and e.g., any program constant. Hence, the adversary is capable of continuously observing the control flow.

The goal of the adversary is to learn additional information about the initial state $i \in I$, i.e., which input variable is assigned which value. Since the adversary knows the value of the instruction pointer at any time, we do not model it explicitly in the following. Instead, we will assume that a state $s \in S$ only consists of a map assigning variables to values.

C. Information Flow Policy Compliance

In accordance with our adversary model, an attacker may learn information about the program state from expressions determining control flow decisions. To prevent sensitive data from being revealed, it is thus desirable to ensure that such data is either not used in a control flow decision or the adversarial information flow caused by the control flow decision does not exceed a certain user-defined threshold.

In the following, we define *information flow policies* allowing these requirements to be expressed on a per-variable basis. Our goal is to verify that a given program P complies with a given information flow policy. If P is non-compliant, our goal is to transform P into a semantically equivalent program P' that is compliant.

Definition 1 (Quantitative Information Flow Policy). *Let P be a program and V the set of variables in P. A* quantitative information flow policy *Ψ is a map assigning each program variable a numeric upper bound on the adversarial information flow when executing P. Formally,*

$$\Psi : V \to \mathbb{R}_{\geq 0} \cup \{\infty\}$$

where we use ∞ to denote no upper bound on the adversarial information flow.

Definition 2 (Policy-Compliance). *Let P be a program with variables V and let Ψ a quantitative information flow policy for P. We say that P* is compliant with *Ψ iff for each variable $v \in V$ the adversarial information flow for v when executing P does not exceed $\Psi(v)$. If P is not compliant with Ψ, we say that P* violates *Ψ.*

Definition 3 (Control Flow Removal Algorithm). *A* Control Flow Removal Algorithm

$$\mathcal{T} : \mathcal{P} \times \{0, 1\}^{|T|} \to \mathcal{P}$$

takes as input a program P and a binary vector t specifying for each control flow transition $\tau_i \in T$ whether it may be revealed *($t_i = 1$) or must be* hidden *($t_i = 0$). It outputs a semantically equivalent program variant P' only containing control flow transitions τ_i where $t_i = 1$. We require that $\mathcal{T}$ does not introduce any new control flow transitions.*

D. Security-Performance Trade-off Analysis

Two fundamental requirements for an analysis of the trade-off between security and performance of a policy-compliant program are a function measuring the security of a given program and a function measuring the performance of a program. We model both as cost functions, i.e., in a *the lower the better* fashion.

Definition 4 (Program Security Measure). *A* program security measure *is a function assigning a non-negative numeric value to a given program.*

$$\mu_s : \mathcal{P} \to \mathbb{R}_{\geq 0}$$

Definition 5 (Program Performance Measure). *A* program performance measure *is a function assigning a non-negative numeric value to a given program.*

$$\mu_p : \mathcal{P} \to \mathbb{R}_{\geq 0}$$

Let $f = (f_1, \ldots, f_n)$ be the objective function of an n-objective optimization (minimization) problem, W the search space and X the set of feasible points of that optimization problem.

Definition 6 (Dominance). *Let f and W as before and $x, y \in W$. We say x* dominates *y or $x \succ y$, iff $\forall i \in [1, n] : f_i(x) \leq f_i(y)$ and $\exists i \in [1, n] : f_i(x) < f_i(y)$.*

Definition 7 (Pareto-optimality). *Let f and X as before. A point $x \in X$ is called* Pareto-optimal with respect to *f iff $\nexists y \in X : y \succ x$. If f is clear from context, we simply call x* Pareto-optimal*. The set of all Pareto-optimal points is called the* Pareto front.

Definition 8 (Non-dominated set). *Let X as before. A set $Y \subseteq X$ is called* non-dominated *iff $\nexists x, y \in Y : x \succ y$.*

Definition 9 (Policy-Aware Security-Performance Trade-off Analysis). *Let P be a program and Ψ a quantitative information flow policy for P. Furthermore, let μ_s be a program*

security measure, μ_p a program performance measure and $\mathcal{T}$ a control flow removal algorithm.

The problem of policy-aware security and performance trade-off (PASAPTO) analysis *is to produce – in expected polynomial time – a set of programs $\mathbb{P}$ such that each program $P' \in \mathbb{P}$ is (i) a $\mathcal{T}$-transformation of P, that is $P' = \mathcal{T}(P, t)$ for some binary vector $t \in \{0, 1\}^{|T|}$, (ii) compliant with policy Ψ and (iii) Pareto-optimal with respect to (μ_s, μ_p).*

E. Notation

We write $x \leftarrow_{\$} X$ to sample x uniformly at random from a set X. $|X|$ denotes the cardinality of a set X. We assume the base of the logarithmic function to be 2.

We denote the i-th unit vector by e_i. By $0_{m,n}$ we denote the $m \times n$ matrix with all entries equal to zero. For a vector v we write v_i to select the i-th component of v and for a matrix A we write $A_{i,:}$ to select the i-th row and $A_{:,j}$ to select the j-th column. For two matrices $A, B \in \mathbb{R}^{m \times n}$ we write $A \circ B$ to denote the Hadamard product of A and B, i.e., the componentwise multiplication.

III. Quantifying Control Flow Leakage

In this section, we present two program security measures (instances of μ_s) capturing the adversarial information flow resulting from the observation of control flow decisions.

To evaluate the security of programs, we rely on established quantitative information flow (QIF) techniques and information theory [6], [29], [41], [51]. Our QIF analysis is decomposed into two steps: an *algebraic interpretation* followed by a *numerical evaluation*. First, we capture the view of any adversary as an equivalence relation on the initial states of a program. Then, we quantitize the equivalence relation to arrive at a numeric value expressing the adversarial information flow when executing the program. The primary benefit of a two-step QIF analysis is that each of the two steps can be considered independently.

A. Algebraic Interpretation

We model the information flow to an observer resulting from an execution of a program as an equivalence relation R on its initial states. $R \subseteq I \times I$ relates two states if an observer cannot distinguish between them. R is called *indistinguishability relation* and induces an *indistinguishability partition* $I_{/R}$ on the set of all possible initial states I.

In the extreme case of $R = I \times I$, the observer cannot distinguish between any of the program's states and has learned nothing from its observation. If, on the other hand, the equivalence classes are singleton sets, the observer has perfect knowledge of the program's initial state. Intuitively, the higher the number of equivalence classes and the smaller the classes, the more information is revealed to the attacker.

We use symbolic execution to compute the indistinguishability partition. Symbolic execution [28] is a method of program evaluation using symbolic variables and expressions rather than actual input values. We obtain symbolic program paths which aggregate input values leading to the same control flow. The resulting equivalence classes contain input values indistinguishable from the point of view of an adversary. We assume the existence of a function Π mapping a program $P = (S, Tr, I, F)$ to its indistinguishability partition, i.e., $\Pi(P) := I/_R$, using symbolic execution.

B. Numerical Evaluation

For the second step, a multitude of valuations have been proposed in the literature [6], [29], [41]. It should be noted that neither of the established definitions is superior to any other definition in all cases. The valuation to use depends on the system model, the adversary model and the question that is supposed to be answered.

We are concerned with the question "How much information can an attacker gain from observing the system?" [41] and define two security metrics measuring the answer in bits. We measure the uncertainty of an attacker about the initial state of a program using the information-theoretic concept of entropy. The adversarial information flow (leakage) is then defined as the reduction in uncertainty before and after observing the program's control flow.

1) Probability and Information Theory:

Definition 10 (Shannon Entropy)**.** *Let X be a discrete random variable with possible values $E = \{x_1, \dots, x_n\}$ and probability mass function $p_X(x)$. The* Shannon entropy *(or* entropy*) is defined as*

$$H(X) := -\sum_{i=1}^{n} p_X(x_i) \log p_X(x_i).$$

Definition 11 (Conditional Shannon Entropy)**.** *Let X and Y be discrete random variables with probability mass functions $p_X(x)$ and $p_Y(y)$. The* conditional Shannon entropy of X given Y *is defined as*

$$H(X|Y) := -\sum_{x \in \mathcal{X}, y \in \mathcal{Y}} p_{X,Y}(x, y) \log \frac{p_{X,Y}(x, y)}{p_Y(y)},$$

where $\mathcal{X}$ and $\mathcal{Y}$ denote the support sets of X and Y and $p_{X,Y}$ the joint probability mass function of X and Y.

Definition 12 (Conditional Minimal Entropy)**.** *Let X and Y be discrete random variables with probability mass functions $p_X(x)$ and $p_Y(y)$. The* conditional minimal entropy of X given Y *is defined as*

$$H_\infty(X|Y) := \min_{y \in \mathcal{Y}} H(X|Y = y),$$

where $\mathcal{Y}$ denotes the support set of Y.

2) Two Information Flow Measures: Let $P = (S, Tr, I, F)$ be a program and R the indistinguishability relation after observing the control flow. We define two discrete random variables [6]: $\mathcal{U} : I \to I$ with probability distribution $p : I \to \mathbb{R}$ represents the distribution of the initial states. $\mathcal{V}_R : I \to I/_R$ maps each initial state to its equivalence class according to R.

Definition 13 (Average Information Flow)**.** *Let P, $\mathcal{U}$, $\mathcal{V}_R$ be as before. We define the* Average Information Flow *of P as*

$$\bar{\mu}(P) := H(\mathcal{U}) - H(\mathcal{U}|\mathcal{V}_R).$$

Definition 14 (Maximum Information Flow)**.** *Let P, $\mathcal{U}$, $\mathcal{V}_R$ be as before. We define the* Maximum Information Flow *of P as*

$$\mu_{max}(P) := H(\mathcal{U}) - H_\infty(\mathcal{U}|\mathcal{V}_R).$$

The *average information flow* measures the leakage of a program using weighted averaging over all equivalence classes. The *maximum information flow* only considers the smallest equivalence class, thus it measures the worst-case leakage of a program. An example illustrating average and maximum information flow can be found in Appendix A-A.

IV. Determining and Establishing Policy-Compliance

In this section, we first show how to quantify variable-specific leakage in order to support quantitative information flow policies. Then, we describe how a program P violating a given quantitative information flow policy Ψ can be transformed into a program variant P' complying with Ψ.

As defined in Definition 1, we consider a program P compliant with a quantitative information flow policy Ψ iff the adversarial information flow of P does not exceed $\Psi(v)$ for any variable v. Formally:

$$P \text{ is compliant with } \Psi \Leftrightarrow \forall v \in V : \mu_{var}(P, v) \le \Psi(v)$$

$\mu_{var}(P, v)$ refers to the maximum information flow of variable v when executing P. Since $\Psi(v)$ defines an upper bound on the adversarial information flow for variable v, we must use a worst-case information flow measure. In the following, we construct this measure based on our results from the previous section.

Consider random variable $\mathcal{U}$ which distinguishes the initial states from each other. To define a variable-specific measure for variable v_i, we want to distinguish only states with differing values of variable v_i. To this end, we consider a partition on the initial states fulfilling the following relation. Two states are equivalent iff the variable assignments of the i-th variable are equal. We denote this partition by $I_{/v_i}$ and define $\mathcal{U}_{|v_i} : I \to I_{/v_i}$. Note that $\mathcal{U}_{|v_i}$ is a random variable as well. We can now define a variable-specific information flow measure.

Definition 15 (Variable-Specific Information Flow)**.** *Let P, $\mathcal{U}$, $\mathcal{V}_R$ be as defined in III-B2 and let $v_i \in V$ be a variable of P. We define the* Variable-Specific Information Flow *of v_i as*

$$\mu_{var}(P, v_i) := H(\mathcal{U}_{|v_i}) - H_\infty(\mathcal{U}_{|v_i}|\mathcal{V}_R).$$

An example of variable-specific information flow is presented in Appendix A-B.

If a program P violates a policy Ψ, then there exists some $v \in V$ such that the adversarial information flow exceeds the defined threshold $\Psi(v)$. In order for P' to be compliant with Ψ, the adversarial information flow with regards to v must be decreased. Adversarial information flow results from the adversary's ability to observe the control flow of a program. Hence, adversarial information flow with regards to a variable v can be reduced by removing control flow statements involving data from v. To this end, a control flow removal algorithm $\mathcal{T}$ as defined in Definition 3 can be applied. Note that policy-compliance can always be established since $\mathcal{T}$ can be used to hide all control flow transitions of a program.

V. Policy-Aware Security and Performance Trade-off Analysis

In this section, we first formalize the problem of policy-aware security and performance trade-off (PASAPTO) analysis as an optimization problem. We then define a decision problem based on the optimization problem and show its NP-hardness. Finally, we present a greedy heuristic and a genetic algorithm efficiently approximating a solution to the optimization problem.

Given a control flow removal algorithm $\mathcal{T}$, besides establishing policy-compliance, we can also explore the trade-off between security and performance (see Definition 9). More precisely, each $t \in \{0,1\}^{|T|}$ can be interpreted as a specific selection of a trade-off. The problem of finding those transitions corresponding to a policy-compliant program with optimal security and performance can be expressed as an optimization problem with objective function

$$f(t) := \begin{pmatrix} \mu_s(\mathcal{T}(P,t)) \\ \mu_p(\mathcal{T}(P,t)) \end{pmatrix} .$$

In more detail, we are interested in the argument minimum for the cost function $f(t)$ such that the program $\mathcal{T}(P,t)$ complies with Ψ. Formally:

$$\begin{aligned} \arg\min_t \quad & f(t) \\ \text{s.t.} \quad & \mathcal{T}(P,t) \text{ is compliant with } \Psi, \\ & t \in \{0,1\}^{|T|} \end{aligned} \tag{1}$$

Since f has multiple objectives, solving this optimization problem will in general not yield a single optimal function argument, but a set of Pareto-optimal solutions.

To analyze the complexity of the problem, we define its corresponding decision problem and a class of security and performance functions which allow to construct programs with certain behaviour.

Definition 16. *Let μ_{var} be a variable-specific information flow measure and μ_p be a program performance measure. We call (μ_{var}, μ_p) polynomial-time linear expandable iff:*

1) *$\forall P \in \mathcal{P}\ \forall n \in \mathbb{N}_0\ \forall v \in V$: we can modify P (in polynomial time) to P' by adding a control flow decision τ_i such that for $t_i = 1 : \mu_{var}(P', v) = \mu_{var}(P, v) + n$ and $\forall v' \in V, v' \neq v : \mu_{var}(P', v') = \mu_{var}(P, v')$.*
2) *$\forall P \in \mathcal{P}\ \forall n \in \mathbb{N}_0\ \forall \tau_i \in T$: we can modify (in polynomial time) P to P' such that for $t_i = 1 : \mu_p(P') = \mu_p(P) + c$ and for $t_i = 0 : \mu_p(P') = \mu_p(P) + 2c$.*
3) *For the empty program P_0 it applies that $\mu_p(P_0) = 0$ and $\forall v \in V : \mu_{var}(P_0, v) = 0$.*

Definition 17 (PASAPTO Decision Problem)**.** *Given* P, Ψ, μ_{var}, μ_s, μ_p *and* $k_s, k_p \in \mathbb{N}_0$, *the* PASAPTO decision problem *is to decide whether* $\exists t \in \{0,1\}^{|T|} : \mathcal{T}(P,t)$ *is compliant to* $\Psi \wedge \mu_s(\mathcal{T}(P,t)) \leq k_s \wedge \mu_p(\mathcal{T}(P,t)) \leq k_p$.

Theorem 1. *If* (μ_{var}, μ_p) *are polynomial-time linear expandable, then the PASAPTO decision problem is NP-hard.*

Proof Sketch. We show a reduction from the 0-1 integer linear programming (0-1-ILP) decision problem, which is known to be a NP-complete. Thereby, every 0-1-ILP variable represents a control flow decision and every 0-1-ILP condition represents a variable in the program. If a control flow decision is revealed, the leakage of each variable shall be increased by the corresponding coefficient in the 0-1-ILP conditions. Thus, the policy represents the conditions of the 0-1-ILP and the performance represents the objective function. The full proof can be found in Appendix B.

Corollary 1.1. *If* μ_{var}, μ_s *and* μ_p *can be computed in polynomial time (e.g., constant time), then the PASAPTO decision problem is NP-complete.*

To overcome the time complexity implications of Theorem 1, we can employ heuristic approaches to efficiently find satisfactory solutions. A heuristic may not necessarily find optimal solutions, but we require any returned solution to satisfy the constraints, i.e., to be policy compliant.

A naïve heuristic could assume that removing a control flow transition always decreases the adversarial information flow. We can describe this assumption formally as follows. Let $\leq_p \subseteq \{0,1\}^{|T|} \times \{0,1\}^{|T|}$ be a partial order defined as $t \leq_p t' :\Leftrightarrow \forall i \in [1, |T|] : t_i \leq t'_i$. For a given program P and a security measure μ_s we consider $g(t) := \mu_s(\mathcal{T}(P,t))$ as a function with input $t \in \{0,1\}^{|T|}$. Then, the stated assumption is equal to $g(t)$ being monotonic increasing. We can prove that this assumption does not hold.

Theorem 2. *Let* $\leq_p$ *and* $g(t)$ *be as defined before. Then,* $g(t)$ *is not monotonic increasing in the variable* $t \in \{0,1\}^{|T|}$ *for partial order* $\leq_p$. *Formally,* $t \leq_p t' \nRightarrow g(t) \leq g(t')$.

Proof. See Appendix C for a proof by contradiction. □

Even if there is no general structure, one could find some special cases where the monotony assumption holds. For example, a program without any nested conditions, as well as the restriction of removing only inner conditions of a program with nested ones fulfill this assumption. Our heuristics implicitly take this knowledge into account.

In the remainder of this section, we present a greedy heuristic providing fast convergence and a genetic algorithm providing well distributed solutions.

A. `GreedyPASAPTO`*: A greedy heuristic*

Our first algorithm `GreedyPASAPTO` (Greedy Policy-aware Security and Performance Trade-off Analysis) is a greedy heuristic providing fast convergence. Starting point of this algorithm is a transformation of P not containing any control flow transitions. We call this program *the all-hidden program*. We know that this program is compliant with Ψ, because it does not entail any adversarial information flow at all. Based on the all-hidden program, we iteratively reveal control flow transitions until revealing any other control flow transition would lead to a non-compliant or dominated program. By incrementally revealing control flow transitions, we expect to gradually obtain policy-compliant programs with better performance.

Require: P – Program under inspection
Ψ – Quantitative information flow policy
Ensure: $\mathbb{P}$ – Non-dominated set of program variants of P
1: **procedure** GREEDYPASAPTO(P, Ψ)
2: $T :=$ *filterControlFlow*(P)
3: $V :=$ *computeVariableSpace*(P)
4: $B := \{0_{|T|,1}\}$
5: $\mathbb{P} := \emptyset$
6: **while** $B \neq \emptyset$ **do**
7: $t \leftarrow_{\$} B$
8: $B := \emptyset$
9: **for all** $i \in \{n \in \mathbb{N} \mid 1 \leq n \leq |T|, t_n = 0\}$ **do**
10: $t' := t + e_i$
11: **if** $\forall v \in V : \mu_{var}(\mathcal{T}(P,t'), v) \leq \Psi(v)$ **then**
12: $B := B \cup \{t'\}$
13: **end if**
14: **end for**
15: $\mathbb{P} := \mathbb{P} \cup \{\mathcal{T}(P,t) \mid t \in B\}$
16: $\mathbb{P} :=$ *filterNonDominated*$(\mathbb{P})$
17: **for all** $t' \in \{t \in B \mid \mathcal{T}(P,t) \notin \mathbb{P}\}$ **do**
18: $B := B \setminus \{t'\}$
19: **end for**
20: **end while**
21: **return** $\mathbb{P}$
22: **end procedure**

Figure 2: `GreedyPASAPTO`: Greedy Heuristic

Structure: `GreedyPASAPTO` is structured as follows. It takes as input a program P and a quantitative information flow policy Ψ and outputs a non-dominated set of programs $\mathbb{P}$. In each iteration step, we consider a base program, starting with the all-hidden program in the first step, and a bit vector set B corresponding to programs with one additional control flow transition revealed. We filter any policy-compliant and non-dominated program and add its corresponding bit vector to the current bit vector set B. The algorithm terminates if every program is non-compliant or dominated by a program of the solution set or if there is no more transition to reveal. Each program corresponding to an element of B is added to the solution set. One of these programs is randomly chosen as a base for the next iteration step. Filtering non-dominated programs is achieved by the subroutine *filterNonDominated*$(\cdot)$ that on input a set of programs outputs the maximum subset of non-dominated programs. The details of `GreedyPASAPTO` are presented in Algorithm 2.

Alternative approach: We prefer the approach of starting with the all-hidden program and revealing control flow transitions over the alternative of starting with the all-revealed, i.e., the original, program and removing control flow transitions for two reasons. First, the program our algorithm starts with is guaranteed to be policy-compliant. The alternative approach on the other hand has to somehow establish policy-compliance by investigating other program variants. In doing so, a large number of non-compliant programs may have to investigated. Second, one could think that in the alternative approach removing additional control flow from a policy-compliant program would always lead to another policy-compliant program and hence policy-compliance does not have to be checked again. However, Theorem 2 shows that this is not true in the general case and hence policy-compliance has to be checked for each candidate program.

B. `GeneticPASAPTO`: *A genetic algorithm*

`GeneticPASAPTO` (Genetic Policy-aware Security and Performance Trade-off Analysis) is a heuristic approach for solving optimization problem (1) based on a genetic meta-heuristic. Genetic algorithms do not require any a priori knowledge about the structure of the search space, thus they fit our problem very well. In contrast to our greedy heuristic, a whole set of not necessarily policy-compliant solutions, the so-called population, is considered and used to generate new solutions. `GeneticPASAPTO` selects the fittest individuals, i.e. binary vectors of size $|T|$, from the population according to some fitness function. Based on the selected individuals, by using so-called crossing and mutation, new individuals are generated which replace the least fittest individuals in the population. This procedure is repeated until a sufficiently large amount of non-dominated solutions have been found or a running time bound has been reached.

Our genetic algorithm uses a population size of N determined by the developer. The algorithm outputs for a program P and a quantitative information flow policy Ψ a non-dominated set of policy-compliant programs of size at most N. Since genetic algorithms may converge to one solution, to obtain a wide selection of solutions for the developer, `GeneticPASAPTO` uses niching methods [40]. The details of `GeneticPASAPTO` are presented in Appendix D.

Fitness function: Our fitness function F is based on a ranking [21] which takes policy-compliance into account. To an individual i we assign $F_i := N - k$ if it is dominated by k individuals in the current population. If a program is not policy-compliant, we assign $F_i := 0$ to penalize such solutions and prefer complying programs.

Crossing and Mutation: In the context of genetic algorithms each component of an individual is called a gene. We cross two individuals by switching the first half of the genes of the parents. For those individuals, mutation is applied with a probability of $\frac{1}{|T|}$ for each gene, i.e. the gene is inverted.

Niching: We choose Sharing [24, pp. 41-49.], [27] as our niching method, because it is recommended for multi-objective optimization [40, p. 84]. If two individuals of a population are in the same niche, i.e., their distance is below a certain threshold σ called the sharing parameter, their fitness is shared. The sharing parameter σ should be chosen carefully. An estimation of a good selection of σ based on the bounds of the search space has been made by Fonseca and Fleming [21]. Applied to our problem, we obtain the unambiguous solution

$$\sigma = \frac{M_1 + M_2 - (m_1 + m_2)}{N - 1},$$

We approximate the bounds of the search space using the properties of two well-known programs. We expect the all-revealed program to have high leakage but good performance. On the other hand, we expect the all-hidden program to have no leakage but bad performance.

By determining the distance of two points, we do not want to weight the influence of one dimension over another, because they may have different size scales. To this end, we standardize both dimensions with respect to the maximum values of the programs above. This leads to the following choice of parameters:

$$M_1 := \frac{\mu_s(P)}{\mu_s(P)} = 1 \qquad m_1 := \frac{\mu_s(\hat{P})}{\mu_s(P)} = 0$$

$$M_2 := \frac{\mu_p(\hat{P})}{\mu_p(\hat{P})} = 1 \qquad m_2 := \frac{\mu_p(P)}{\mu_p(\hat{P})},$$

where $\hat{P} := \mathcal{T}(P, (0, \ldots, 0)^T)$ represents the all-hidden program. In the following, we denote the scale factor by

$$S := \left(\frac{1}{\mu_s(P)}, \frac{1}{\mu_p(\hat{P})}\right)^T.$$

The fitness is now shared with other individuals in the same niche. We define a sharing function [24, p. 45],

$$sh : [0, \infty) \to [0, 1]$$

with

$$sh(d) := \begin{cases} 1 - \frac{d}{\sigma}, & \text{if } d < \sigma \\ 0, & \text{otherwise} \end{cases}$$

where d describes a metric of two points in the search space. We use the Euclidean metric in the following.

Based on the simple fitness function F we can now define a shared fitness function F'. F' takes as arguments an individual i and a matrix M which contains the individuals of a population as columns.

$$F'(i, M) := \frac{F_i}{\sum_{j=1}^{N} sh(d(f(M_{:,i}) \circ S, f(M_{:,j}) \circ S))}$$

Convergence: Based on the shared fitness function the evolution process is repeated until the maximum number of iterations (user-defined parameter X) is reached or the convergence criterion is fulfilled. We use Maximum Allowable Pareto Percentage as our convergence criterion, i.e., the algorithm terminates if the percentage of non-dominated individuals in the current population exceeds the user-defined threshold α. By default, we use $\alpha = 0.7$. The quality of the solution set depends on carefully chosen parameters α and X.

VI. Application to Secure Cloud Computing

In this section, we construct a PASAPTO analysis for Dataflow Authentication (DFAuth) [20], an approach for outsourcing computations.

A. Dataflow Authentication

DFAuth computes on encrypted data in the cloud using a combination of homomorphic encryption and a Trusted Module (TM), e.g., an Intel SGX enclave [3], [26], [43]. Compared to a solution relying solely on SGX, DFAuth offers a number of advantages. Since software vulnerabilities in the enclave can be used to disarm the security guarantees of SGX [34] and the number of vulnerabilities scales with the size of the code base, it is advisable to keep the trusted code base (TCB) in the enclave as small as possible. DFAuth minimizes the exposure to vulnerabilities by providing a small and program-independent TCB, which can be reused across applications. Application vulnerabilities are confined to the untrusted program computing on encrypted data. Due to its small size, the TCB in the TM can be hardened similar to implementations of cryptographic primitives. For example, the TM implementation can be made branchless, heapless and constant-time.

System Overview: DFAuth [20] considers a scenario between a trusted client and an untrusted cloud server, which is equipped with a Trusted Module (TM). The client wishes to execute a program on sensitive data at the cloud server.

In the setup phase, the client transforms the intended program using a DFAuth-enabled compiler. The client transfers the transformed program to the server and (securely) deploys program metadata into the TM. In the runtime phase, the client encrypts its inputs before transferring them to the server. The server executes the program and returns an encrypted result to the client. On the server, DFAuth performs computations efficiently in unprotected memory using partially-homomorphic encryption (PHE). Conversions between incompatible PHE schemes and comparisons for control-flow decisions are performed through TM invocations. To allow efficient program execution (cf. Introduction), DFAuth does not protect the control flow of the executed program.

Adversary Model: DFAuth assumes an active adversary controlling the cloud server [20], who can (i) read and modify the contents of all variables and the program text (except in the TM) (ii) observe and modify the control flow (except in the TM) (iii) do all of this arbitrarily interleaved.

The security guarantee of DFAuth is to reveal only the information about the inputs to the untrusted cloud server that can be inferred from the program's executed control flow. DFAuth reduces an active to a passive adversary by preventing the adversary from learning any information beyond the intended information flow of the program.

B. A PASAPTO Analysis for DFAuth

Inherently, DFAuth considers an active adversary, but due to its security guarantees any adversary is limited to passively observing the control flow of the program executed on the cloud server. Recall from Section II-B that in this work we consider a passive adversary capable of continuously observing the control flow of an execution of a program. As such, we can extend DFAuth with a PASAPTO analysis to further reduce the adversarial information flow. This combination forms the basis of our implementation of a Dataflow Authentication Trade-off Analyzer (`DFATA`), which is explained in more detail in the next section.

Security and Performance Measures: In Section III we presented two program security measures capturing the leakage resulting from control flow observations. Both, average information flow ($\bar{\mu}$) and maximum information flow (μ_{max}), can be used to instantiate the program security measure μ_s. To determine compliance of a program with a quantitative information flow policy, we apply the variable-specific information flow measure (μ_{var}) introduced in Section IV.

We consider two instantiations of the program performance measure μ_p. The running time performance measure $\hat{\mu}$ captures the elapsed wall-clock time of the execution of a DFAuth-transformed program. An alternative program performance measure can be defined by applying program analysis techniques to count the instructions used by the program and assign each DFAuth operation a cost value. However, this measure is outside of the scope of this work.

Control Flow Removal Algorithm: We consider two instantiations of $\mathcal{T}$ which we refer to as straightlining ($\mathcal{T}_s$) and oblivious state update ($\mathcal{T}_o$). $\mathcal{T}_s$ rewrites code fragments containing conditional instructions into semantically equivalent code not containing any control flow transitions. For example, bitwise operations can be combined with a constant-time conditional assignment operation to avoid conditional instructions [44]. $\mathcal{T}_o$ executes both the *then* and the *else* branch of a conditional [1], [48]. $\mathcal{T}_o$ forks the state of the program before branching and obliviously updates the program state with the correct one. To this end, we extend DFAuth with an additional TM invocation. Provided with the two state variants and the conditional, the TM determines the correct variant and returns it to the cloud server. Before returning, the correct variant is re-randomized such that the untrusted part of the server remains oblivious as to which of the two states was returned. Note that this extension is in line with DFAuth's goal of a program-independent trusted code base [20].

VII. Implementation

In this section, we present details about `DFATA`, our Java-based Dataflow Authentication Trade-off Analyzer. At the core of our implementation is Soot, a framework for analyzing and manipulating Java programs [33]. Soot is used to implement our own DFAuth compiler, detection of control flow leaks and control flow removal. We implement the DFAuth TM in an SGX enclave.

We compute the indistinguishability partition required to quantify control flow leakage based on JBSE (Java Bytecode Symbolic Executor) [10], [11]. Using symbolic execution we obtain the set of symbolic paths of a program and the corresponding path conditions. By construction, the path

conditions are equal to the constraints of each equivalence class. We assume the random variable $\mathcal{U}$ to be distributed uniformly and transform the constraints into a system of linear inequalities for each equivalence class. The numeric evaluation of each equivalence class is determined using LattE [39]. LattE implements Barvinok's algorithm [7] to compute the size of the equivalence classes exactly and in polynomial time.

`DFATA` supports our two heuristics as well as an exhaustive search trade-off analysis algorithm. `DFATA` takes Java Bytecode operating on plaintext data and an associated information flow policy as input. It outputs the result of the trade-off analysis as a set of fully executable programs operating on encrypted data using DFAuth.

`DFATA` currently operates only on a subset of Java. It performs intra-procedural analysis and does not handle exceptions, because it is limited by its tools – DFAuth and the QIF analysis. Extensions are possible, but orthogonal to our work on heuristics. For an overview on techniques, tools and trade-offs for model counting involving non-linear numeric constraints, we refer to a survey by Borges et al. [9]. We refer to Aydin et al. [5] for how to perform model counting on string (sequences of characters) constraints and mixed string and integer constraints. For multi-instantiation of a class in a sensitive and an insensitive context, we refer to Dong et al. [18]. Also, the current implementation is not optimized for performance, but for the evaluation of the quality of our heuristics.

VIII. Evaluation

In this section, we present the results collected in two experiments in which we applied `DFATA`. In the first experiment, we consider an implementation of an electronic auction with sealed bids. In the second experiment, we inspect a program performing decision tree evaluation on medical data. We chose these programs such that we are able to explore all program variants in order to evaluate the effectiveness of our heuristics, but complex enough to be non-trivial in the context of computation on encrypted data (e.g., FHE, MPC).

In each experiment, we first use `DFATA` in exhaustive mode to obtain all program variants as well as their average information flow ($\bar{\mu}$) and running time performance ($\hat{\mu}$). We determine $\hat{\mu}$ by executing each variant multiple times on random inputs, then computing the mean over all measurements. From these results, we determine the Pareto front of the PASAPTO optimization problem. We then execute each of our two probabilistic heuristics multiple times.

In each run, we compute the Hypervolume Indicator (HVI) [58] of the heuristic solution set with respect to the dimensions $\bar{\mu}$ and $\hat{\mu}$. The HVI is an established measure used to determine the quality of multi-objective solution sets [4], [49]. We evaluate the quality of our algorithms using the relative difference between the HVI of a heuristic solution set and the HVI of the Pareto front:

$$\mathrm{HVI}_{\mathrm{rel}} := \frac{\mathrm{HVI}_{\mathrm{heuristic}}}{\mathrm{HVI}_{\mathrm{Pareto\ front}}} - 1.$$

Elements of high quality solution sets according to $\mathrm{HVI}_{\mathrm{rel}}$ are close to the Pareto front and cover a wide range of trade-offs.

For each experiment and heuristic, we perform 101 runs. We perform an odd number of runs such that we can unambiguously identify a single run as the *median run* according to $\mathrm{HVI}_{\mathrm{rel}}$. Scatter plots presented in the following show data points for this particular run. In addition to the median $\mathrm{HVI}_{\mathrm{rel}}$, we present the 95th percentile ($Q_{0.95}$) of $\mathrm{HVI}_{\mathrm{rel}}$ over all runs.

Because our implementation of `DFATA` is not optimized for performance, but for the evaluation of the quality of the heuristics, we cannot provide wall-clock running times of our heuristics. However, we can estimate the total running time t of a heuristic by $t = v * (s + p) + h$ where v refers to the number of program variants visited, s and p denote the time to compute $\bar{\mu}$ respectively $\hat{\mu}$ of a program variant, and h refers to the running time of the heuristic itself.

All experiments were conducted in the Microsoft Azure Cloud using Azure Confidential Computing VM instances of type `Standard_DC4s`. Each instance runs Ubuntu Linux 18.04 and has access to 4 cores of an SGX-capable Intel Xeon E-2176G CPU and 16 GiB RAM.

A. Electronic Sealed-Bid Auction

In a sealed-bid auction, individual bids $\{h_0, \ldots, h_{n-1}\}$ must be kept confidential. The winner of the auction is identified by the highest bid and is announced publicly.

Experimental Setup: Consider the following pseudocode implementing such an auction.

```
1 int auction(int[] h)
2   int l = 0;
3   for (int i = 0; i < h.length; i++)
4     if (h[i] > h[l])
5       l = i;
6   return l;
```

On input an array h containing n bids, the program determines the identity of the winner l as the array index of the winning bid. The input array h is ordered randomly to protect the identities of the bidders. In doing so, we also ensure a fair determination of the winner in case the highest bid does not unambiguously identify a winner.

In this experiment, we assume $n = 10$. After loop unrolling, the resulting program performs $n - 1$ comparisons, respectively control flow transition. Since one can decide whether to *reveal* or *hide* for each transition, our algorithms operate on a search space containing $2^{n-1} = 512$ program variants. For `GeneticPASAPTO` we use a population size of $N := 2 \cdot |T| = 19$ and a bound of $X := 10 \cdot |T| = 90$.

Evaluation Results: Figure 3 shows the median leakage and running time of all possible program variants grouped by their number of hidden control flow transitions. The number of variants per aggregation group is $\binom{n-1}{k}$ where k is the number of hidden control flow transitions. For example, the search space contains $\binom{9}{2} = 36$ variants with 2 hidden transitions. The chart experimentally confirms the negative correlation between leakage and running time we expect.

Figure 4 relates the behaviour of the median run of `GreedyPASAPTO` to the entire search space. Figure 5 does

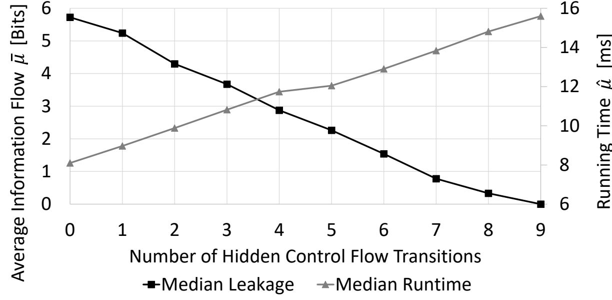


Figure 3: Median leakage (left y-axis) and median running time (right y-axis) of the set of all Auction program variants grouped by number of hidden control flow transitions.

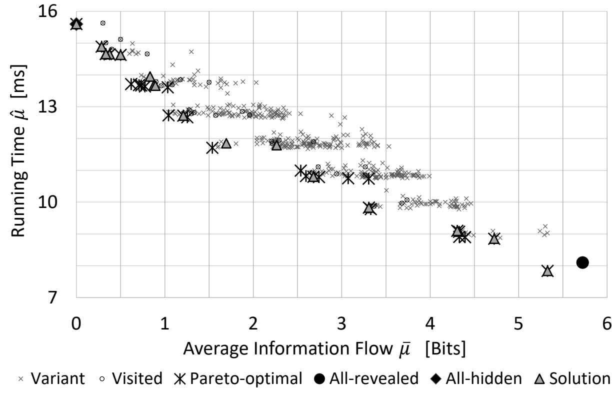


Figure 4: Greedy algorithm applied to Auction Program.

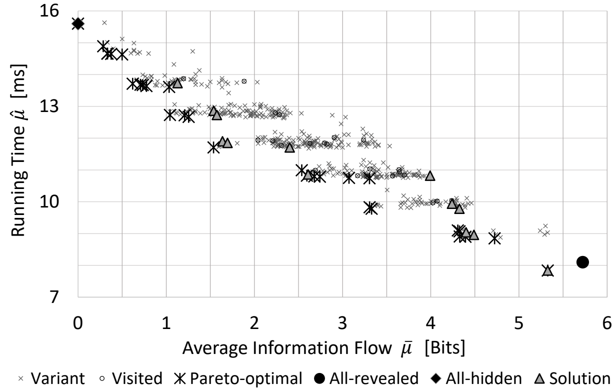


Figure 5: Genetic algorithm applied to Auction Program

the same for `GeneticPASAPTO`. Each plot contains the entire search space of 512 program variants. The distinguished all-hidden and all-revealed programs are highlighted. The set of Pareto-optimal programs is a subset of the search space and contains elements of the Pareto front of the PASAPTO optimization problem. The set of *visited* programs is a subset of the search space and contains programs which have been investigated by the algorithm. Points marked as *solution* are part of the solution set output by the algorithm.

As is expected, no program with worse performance than the all-hidden program exists. One would expect the all-revealed program to provide the best performance, but `DFATA` found another one with better performance. However, we believe this to be only due to the inaccurate nature of the running time measurement. The performance range over all program variants reaches from 7.84 ms for the best performing program to 15.63 ms for the all-hidden program. The security range reaches from a leakage of 0 bits for the all-hidden program to 5.723 bits for the all-revealed program.

In Figure 4 we can see that `GreedyPASAPTO` visited only 46 points and output 18 points in the solution set. The algorithm still found a solution close to the Pareto front. In the median, the heuristic solution is $\mathrm{HVI}_{\mathrm{rel}} = 5.35\%$ ($Q_{0.95} = 19.18\%$) worse than the Pareto front.

In Figure 5 we can see that `GeneticPASAPTO` also produced a solution close to the Pareto front, but visited 38 programs and output 13 in the solution set. For the genetic algorithm, we obtain $\mathrm{HVI}_{\mathrm{rel}} = 19.47\%$ with $Q_{0.95} = 30.36\%$. Even if the quality of the median run is worse than the quality of `GreedyPASAPTO`, we can repeat the execution of `GeneticPASAPTO` to obtain a higher quality since the algorithm is probabilistic. Moreover, the advantage of `GeneticPASAPTO` is that it can potentially find any solution, whereas `GreedyPASAPTO` could miss some solutions in every run by structure.

In this experiment, measuring the QIF of a program variant approximately takes time $s = 26\,\mathrm{s}$ and measuring the performance of a program variant approximately takes time $p = 12\,\mathrm{ms}$. In conclusion, the running time would be roughly $t = 1200\,\mathrm{s}$ for `GreedyPASAPTO` ($h = 0.1\,\mathrm{ms}$), and $t = 1000\,\mathrm{s}$ for `GeneticPASAPTO` ($h = 6\,\mathrm{ms}$).

B. Decision Tree Evaluation

In this experiment, we use `DFATA` on a program performing decision tree (DT) evaluation on sensitive medical data. Consider the use case of a research institution providing a DT evaluation service to medical institutions such as hospitals. Medical institutions can submit patient's health data and obtain the result of the DT classification to aid their diagnosis. Since patient data is highly confidential, the research institution has to compute on encrypted data using DFAuth.

The security goal is to protect the sensitive inputs to the decision tree. The sensitivity of each input is provided as a quantitative information flow policy. The protection of the output of the decision tree is not a security goal and may be learned by the DT service provider.

Experimental Setup: We consider a program making predictions about breast cancer based on the decision tree by Sumbaly et al. [52]. Provided with medical information, the program outputs a prediction of whether or not the patient has breast cancer. The pruned decision tree takes six input variables and performs 13 control flow decisions. Thus, the size of the search space is $2^{13} = 8192$. The attribute represented by each variable is described in Table I.

Table I: Overview on variables and our QIF policy

Variable	Attribute	$\Psi(v_i)$
v_1	Clump Thickness	2
v_2	Uniformity Cell Size	∞
v_3	Uniformity Cell Shape	∞
v_4	Marginal Adhesion	1
v_5	Bare Nuclei	∞
v_6	Bland Chromatin	0

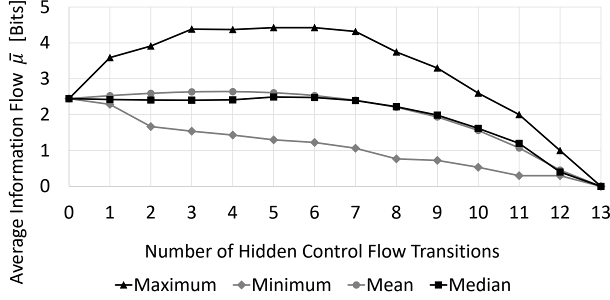


Figure 6: Median leakage of the set of all Decision Tree program variants grouped by number of hidden control flow transitions.

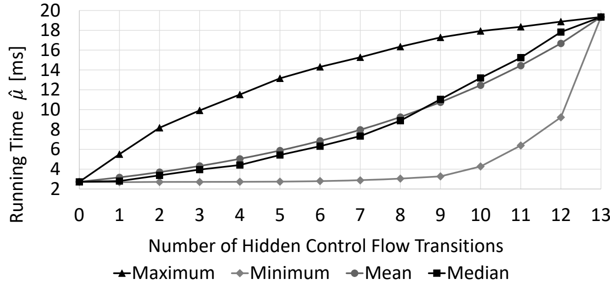


Figure 7: Median running time of the set of all Decision Tree program variants grouped by number of hidden control flow transitions.

Input variables are integers in the interval $[1, 10]$, which our leakage model captures accordingly. The information of each variable is $\log(10) = 3.32$. Our analysis can take into account that not all variables are equally sensitive. Under the assumption that (i) the Bland Chromatin attribute is critically sensitive and no information about it must leak, (ii) the Clump Thickness attribute is highly sensitive, (iii) the Marginal Adhesion attribute is sensitive and (iv) all other variables are not sensitive, a developer may define a quantitative information flow policy as presented in the third column of Table I. We will assume this policy in our experiment. For `GeneticPASAPTO` we use a population size of $N := 4 \cdot |T| = 52$ and a bound of $X := 20 \cdot |T| = 260$.

Evaluation Results: Figures 6 and 7 present results concerning the search space, i.e., all program variants. Figures 8 and 9 present the results of our two heuristic algorithms.

Figure 6 shows leakage depending on the number of hidden control flow transitions. For k hidden transitions, $\binom{13}{k}$ programs are aggregated. In contrast to the previous experiment (see Figure 3), we can see that the median leakage does not decrease before more than 7 transitions are hidden. The mean leakage even increases in comparison to the all-revealed program. This highlights that as per Theorem 2 leakage can increase when removing control flow transitions.

Figure 7 shows the running time depending on the number of hidden transitions. In particular, the minimum of aggregated variants shows that up to 8 transitions can be hidden without significantly increasing the running time. The analysis shows that the leakage and running time of complex programs is not linear in the number of hidden transitions and it is important to find combinations of hidden transitions that barely increase the running time, but significantly reduce leakage.

Figures 8 and 9 present the results of the median runs for each of our two algorithms. Note that for this experiment, we do not show the data points of all 8192 program variants, but only those which are policy-compliant or have been visited by our heuristic. The performance range over all policy-compliant program variants reaches from 4.231 ms for the best performing program to 19.341 ms for the all-hidden program. The security range reaches from a leakage of 0 bits for the all-hidden program to a program variant with 3.358 bits leakage. This is even higher than the leakage of the non-compliant all-revealed program with 2.447 bits and additionally has a worse performance of 7.891 ms compared to 2.720 ms. This experimentally confirms Theorem 2, which says that the adversarial information flow might increase when removing control-flow.

As in the auction experiment, `GreedyPASAPTO` only evaluated a few variants and still produced a solution close to the Pareto front. The algorithm inspected 86 program variants and output 14 programs in the solution set. Visited points that are better than the Pareto front are not compliant with our policy and are not added to the solution set. For this algorithm, we have $\text{HVI}_{\text{rel}} = 1.76\%$ with the 95th percentile $Q_{0.95} = 3.57\%$. The best running time for a policy-compliant program is 4.231 ms, which is also the best running time on the Pareto front.

`GeneticPASAPTO` visited 415 program variants and output 11 programs in the solution set. Recall that a design goal of this algorithm is the diversity of the elements in the solution set, which is implemented using Niching. For example, the solution set of `GreedyPASAPTO` contains clusters of non-dominated solutions, e.g., a lot of program variants with running time between 4 and 5 ms with a large difference in leakage but only little difference in performance. In contrast, Niching avoids clustering and outputs a solution set with sufficiently differing values to simplify the choice of the developer. This can lead to a lower quality solution, because it can cause the fitness of two optimal points to be shared resulting in both being possibly removed from the population. For this algorithm, we have $\text{HVI}_{\text{rel}} = 14.08\%$ with the 95th percentile of $Q_{0.95} = 33.80\%$. The best running time found for a policy-compliant program is 5.353 ms which is 26.5% worse than the best running time on the Pareto front. The second

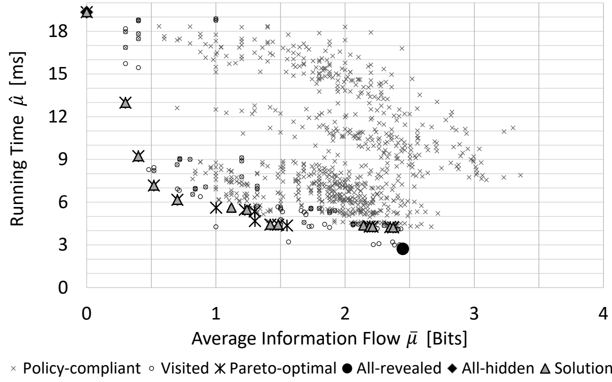


Figure 8: Greedy algorithm applied to Decision Tree Program

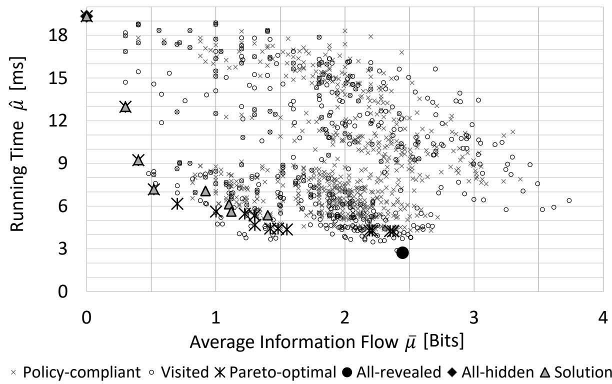


Figure 9: Genetic algorithm applied to Decision Tree Program

experiment provides better solutions than the first one. We believe this is caused by the fact that the second experiment has more structure that can be used by the heuristics.

In this experiment, measuring the QIF of a program variant approximately takes time $s = 15\,\mathrm{s}$ and measuring the performance of a program variant approximately takes time $p = 8\,\mathrm{ms}$. In conclusion, the running time would be roughly $t = 1250\,\mathrm{s}$ for `GreedyPASAPTO` ($h = 0.5\,\mathrm{ms}$), and $t = 6200\,\mathrm{s}$ for `GeneticPASAPTO` ($h = 3\,\mathrm{s}$).

Since the running time is dominated by the QIF analysis, we note that a further, possibly very effective optimization is to update the QIF measure, instead of re-computing it from scratch. We do not have an estimate how effective that optimization would be.

IX. Related Work

Our work is related to the *Trade-off between Security and Performance*, *Language-Based Information Flow Security* and *(Control Flow) Side Channels and Defenses*.

Trade-off between Security and Performance: This trade-off has for example been explored in the context of secured networked control systems [56], cyber-physical systems [57], block ciphers [53], timing attacks in cryptographic code [19] and range queries on encrypted data [36].

Wolter et al. [54] present a generic security-performance trade-off model based on generalised stochastic Petri nets. However, their model assumes that recovery from an insecure system state is possible. We do not think that recovery from adversarial information flows is possible, hence their model cannot be applied in our case.

Language-Based Information Flow Security: For an overview on language-based information flow security, we refer to a survey by Sabelfeld and Myers [50].

The program security measures considered in this work are based on the concept of quantitative information flow (QIF) and inspired by the work of Backes et al. [6]. For foundations of QIF, we refer to Köpf and Basin [31], Smith [51], Heusser and Malacaria [25], Malacaria [41] and Klebanov [29]. Our measures are exact rather than approximate and are based on a two-step QIF analysis consisting of an algebraic interpretation followed by a numeric evaluation. The first step is based on symbolic execution, which has been used previously for information flow analysis [45]–[47]. Our PASAPTO analysis does not mandate a specific program security measure, but can be instanciated using other measures. For example: Leakwatch [14] estimates leakage of Java programs by repeatedly executing them. Malacaria et al. [42] consider an approximation of information flow based on noisy side-channels. Kučera et al. [32] use the concept of an attacker's belief [15], which orthogonally captures the attacker's accuracy besides uncertainty.

Information flow policies and checking for their compliance has been considered in [2], [25], [55]. Policy establishment has for example been considered in [2], [32]. Recently, Kučera et al. [32] presented a program synthesis similar to our program transformation. Their approach also takes a program and a policy as inputs and outputs a policy-compliant program. However, their procedure only applies to probabilistic programs and works by adding uncertainty to the program's output while we consider side-channels and preserve the original program's semantics.

(Control Flow) Side-Channels and Defenses: An approach to avoid a broad class of side-channels is to produce constant-time code not making any control flow decisions on secret data. Molnar et al. [44] present a source-to-source transformation producing such code by relying on bitwise operations and a constant-time conditional assignment operation. Similarly, Cauligi et al. [13] define their own domain specific language to produce code adhering to these requirements. Raccoon [48] transactionally executes both branches of a control flow statement and ensures that the program state is updated obliviously using the correct transaction. GhostRider [37] defends against side-channels based on memory access pattern by obfuscating programs such that their memory access pattern is independent of control flow instructions.

X. Conclusion

We show how to efficiently compute policy-compliant and approximately Pareto-optimal trade-offs between leakage and performance where the decision problem is NP-hard. For a given Java Bytecode program our implementation proposes semantically equivalent, side-channel reduced Java programs for the execution within DFAuth-protected SGX enclaves from which developers can select their desired trade-off. We show the practical feasibility of this approach for computing on encrypted data in a commercial cloud environment using two example programs. The combined protection by DFAuth (security against active attackers and program-independent enclave code) and PASAPTO (performance-optimized side-channel reduction) is more secure against a number of attacks (side-channel analysis, software vulnerability exploitation) than executing the unmodified program in SGX, but orders of magnitude faster than executing the program using fully homomorphic encryption.

Acknowledgments

This work was in part supported by the German Federal Ministry for Economic Affairs and Energy during the TRADE EVs project.

References

[1] J. Agat, "Transforming out timing leaks," in *ACM POPL 2000*.

[2] J. B. Almeida, M. Barbosa, G. Barthe, H. Pacheco, V. Pereira, and B. Portela, "Enforcing ideal-world leakage bounds in real-world secret sharing MPC frameworks," in *IEEE CSF 2018*.

[3] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," in *HASP 2013*.

[4] C. Audet, J. Bigeon, D. Cartier, S. Le Digabel, and L. Salomon, "Performance indicators in multiobjective optimization," *Optimization Online*, 2018.

[5] A. Aydin, W. Eiers, L. Bang, T. Brennan, M. Gavrilov, T. Bultan, and F. Yu, "Parameterized model counting for string and numeric constraints," in *ACM ESEC/FSE 2018*.

[6] M. Backes, B. Köpf, and A. Rybalchenko, "Automatic discovery and quantification of information leaks," in *IEEE S&P 2009*.

[7] A. I. Barvinok, "A polynomial time algorithm for counting integral points in polyhedra when the dimension is fixed," *Math. Oper. Res.*, vol. 19, no. 4, 1994.

[8] D. J. Bernstein, "Curve25519: New diffie-hellman speed records," in *PKC 2006*.

[9] M. Borges, Q. Phan, A. Filieri, and C. S. Pasareanu, "Model-counting approaches for nonlinear numerical constraints," in *NASA NFM 2017*.

[10] P. Braione, G. Denaro, and M. Pezzè, "Enhancing symbolic execution with built-in term rewriting and constrained lazy initialization," in *ACM ESEC/FSE 2013*.

[11] ——, "Symbolic execution of programs with heap inputs," in *ACM ESEC/FSE 2015*.

[12] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *USENIX 2003*.

[13] S. Cauligi, G. Soeller, F. Brown, B. Johannesmeyer, Y. Huang, R. Jhala, and D. Stefan, "Fact: A flexible, constant-time programming language," in *IEEE SecDev 2017*.

[14] T. Chothia, Y. Kawamoto, and C. Novakovic, "Leakwatch: Estimating information leakage from java programs," in *ESORICS 2014*.

[15] M. R. Clarkson, A. C. Myers, and F. B. Schneider, "Belief in information flow," in *IEEE CSFW 2005*.

[16] I. Corporation, "Intel software guard extensions (intel sgx) sdk for linux os," 2019, https://download.01.org/intel-sgx/linux-2.6/docs/Intel_SGX_Developer_Reference_Linux_2.6_Open_Source.pdf.

[17] G. B. Dantzig, "Maximization of a linear function of variables subject to linear inequalities," *Activity analysis of production and allocation*, vol. 13, 1951.

[18] Y. Dong, A. Milanova, and J. Dolby, "Jcrypt: Towards computation over encrypted data," in *PPPJ 2016*.

[19] G. Doychev and B. Köpf, "Rational protection against timing attacks," in *IEEE CSF 2015*.

[20] A. Fischer, B. Fuhry, F. Kerschbaum, and E. Bodden, "Computation on encrypted data using data flow authentication," *PoPETs*, 2020.

[21] C. M. Fonseca and P. J. Fleming, "Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization," in *ICGA 1993*.

[22] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *ACM STOC 2009*.

[23] C. Gentry, S. Halevi, and N. P. Smart, "Homomorphic evaluation of the AES circuit," in *CRYPTO 2012*.

[24] D. E. Goldberg, J. Richardson *et al.*, "Genetic algorithms with sharing for multimodal function optimization," in *Genetic algorithms and their applications: Proceedings of the Second International Conference on Genetic Algorithms*. Hillsdale, NJ: Lawrence Erlbaum, 1987, pp. 41–49.

[25] J. Heusser and P. Malacaria, "Quantifying information leak vulnerabilities," *CoRR*, vol. abs/1007.0918, 2010.

[26] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. del Cuvillo, "Using innovative instructions to create trustworthy software solutions," in *HASP 2013*.

[27] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

[28] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, 1976.

[29] V. Klebanov, "Precise quantitative information flow analysis - a symbolic approach," *Theor. Comput. Sci.*, vol. 538, 2014.

[30] V. Klee and G. J. Minty, "How good is the simplex algorithm," Washington University Seattle Dept. of Mathematics, Tech. Rep., 1970.

[31] B. Köpf and D. A. Basin, "An information-theoretic model for adaptive side-channel attacks," in *ACM CCS 2007*.

[32] M. Kucera, P. Tsankov, T. Gehr, M. Guarnieri, and M. T. Vechev, "Synthesis of probabilistic privacy enforcement," in *ACM CCS 2017*.

[33] P. Lam, E. Bodden, O. Lhotak, and L. Hendren, "The soot framework for java program analysis: a retrospective," in *Cetus Users and Compiler Infastructure Workshop*, ser. CETUS, 2011.

[34] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *USENIX Security 2017*.

[35] 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 *USENIX Security 2017*.

[36] J. Li and E. Omiecinski, "Efficiency and security trade-off in supporting range queries on encrypted databases," in *IFIP 2005*.

[37] C. Liu, A. Harris, M. Maas, M. W. Hicks, M. Tiwari, and E. Shi, "Ghostrider: A hardware-software system for memory trace oblivious computation," in *ASPLOS 2015*.

[38] C. Liu, M. Hicks, and E. Shi, "Memory trace oblivious program execution," in *IEEE CSF 2013*.

[39] J. A. D. Loera, R. Hemmecke, J. Tauzer, and R. Yoshida, "Effective lattice point counting in rational convex polytopes," *J. Symb. Comput.*, vol. 38, no. 4, 2004.

[40] S. W. Mahfoud, "Niching methods for genetic algorithms," Ph.D. dissertation, University of Illinois at Urbana-Champaign, USA, 1996.

[41] P. Malacaria, "Algebraic foundations for information theoretical, probabilistic and guessability measures of information flow," *CoRR*, vol. abs/1101.3453, 2011.

[42] P. Malacaria, M. H. R. Khouzani, C. S. Pasareanu, Q. Phan, and K. S. Luckow, "Symbolic side-channel analysis for probabilistic programs," in *IEEE CSF 2018*.

[43] 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," in *HASP 2013*.

[44] D. Molnar, M. Piotrowski, D. Schultz, and D. A. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *ICISC 2005*.

[45] Y. Noller, R. Kersten, and C. S. Pasareanu, "Badger: complexity analysis with fuzzing and symbolic execution," in *ACM ISSTA 2018*.

[46] J. Obdrzálek and M. Trtík, "Efficient loop navigation for symbolic execution," in *ATVA 2011*.

[47] Q. Phan and P. Malacaria, “Abstract model counting: a novel approach for quantification of information leaks,” in *ACM ASIA CCS 2014*.

[48] A. Rane, C. Lin, and M. Tiwari, “Raccoon: Closing digital side-channels through obfuscated execution,” in *USENIX Security 2015*.

[49] N. Riquelme, C. von Lücken, and B. Barán, “Performance metrics in multi-objective optimization,” in *CLEI 2015*.

[50] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE J. Sel. Areas Commun.*, vol. 21, no. 1, 2003.

[51] G. Smith, “On the foundations of quantitative information flow,” in *FOSSACS 2009*.

[52] R. Sumbaly, N. Vishnusri, and S. Jeyalatha, “Diagnosis of breast cancer using decision tree data mining technique,” *International Journal of Computer Applications*, vol. 98, no. 10, 2014.

[53] S. Wei, J. Wang, R. Yin, and J. Yuan, “Trade-off between security and performance in block ciphered systems with erroneous ciphertexts,” *IEEE Trans. Information Forensics and Security*, vol. 8, no. 4, 2013.

[54] K. Wolter and P. Reinecke, “Performance and security tradeoff,” in *SFM 2010*.

[55] H. Yasuoka and T. Terauchi, “On bounding problems of quantitative information flow,” *Journal of Computer Security*, vol. 19, no. 6, 2011.

[56] W. Zeng and M.-Y. Chow, “A trade-off model for performance and security in secured networked control systems,” in *IEEE ISIE 2011*.

[57] H. Zhang, Y. Shu, P. Cheng, and J. Chen, “Privacy and performance trade-off in cyber-physical systems,” *IEEE Network*, vol. 30, no. 2, 2016.

[58] E. Zitzler and L. Thiele, “Multiobjective optimization using evolutionary algorithms - A comparative case study,” in *PPSN V*, vol. 1498. Springer, 1998.

APPENDIX A
EXAMPLES

A. Average and maximum information flow measures

We illustrate our *adversary model* and the definitions of *indistinguishability partition* and *information flow measures* using the following program P.

```
1  int example1(int x)
2    if (x > 42)
3      x *= 2
4    return x
```

Any possible state of P contains one variable x. To simplify the exposition, let us assume that the domain of x is $D_8 = [-128, 127]$, i.e. x is a signed 8-bit integer. As the adversary knows the program text, the domain of x is known before the execution of the program. Hence the indistinguishability partition contains a single equivalence class.

$$I/_{R_{start}} = \{\{x \in D_8\}\}$$

By observing the control flow – more precisely whether the then-branch (line 3) is taken or not – the attacker can infer whether or not $x > 42$. The indistinguishability partition of the program after observing the control flow (after program execution) is thus

$$I/_R = \{\{x \in D_8 \mid x > 42\}, \{x \in D_8 \mid x \leq 42\}\}.$$

If we assume $\mathcal{U}$ to be distributed uniformly, i.e. each initial state is equally likely, then the initial uncertainty is

$$H(\mathcal{U}) = -\log \frac{1}{2^8} = 8 \text{ bits.}$$

The uncertainty after the control flow observation is given by the conditional entropy. Each equivalence class $E \in I/_R$ occurs with probability $\frac{|E|}{|I|}$, thus we obtain

$$\begin{aligned} H(\mathcal{U}|\mathcal{V}_R) &= -\sum_{E \in I/_R} \frac{|E|}{|I|} \log |E| \\ &= -\frac{85}{256} \log \frac{1}{85} - \frac{171}{256} \log \frac{1}{171} = 7.08 \text{ bits.} \quad (2) \end{aligned}$$

The average information flow of program P is

$$\bar{\mu}(P) = 8 - 7.08 = 0.92 \text{ bits.}$$

In (2) we can see that the then-branch is less likely than the else-branch but leads to a higher amount of information flow. To determine the worst-case leakage we compute the maximum information flow

$$\mu_{max}(P) = H(\mathcal{U}) - H_\infty(\mathcal{U}|\mathcal{V}_R) = 8 - \left(-\log \frac{1}{85}\right) = 1.59 \text{ bits,}$$

which only considers the equivalence class associated with the branch resulting in the largest adversarial information flow.

B. Variable-specific information flow

We consider the following example code of a program P and the information flow policy Ψ with $\Psi(x_1) = 2$ and $\Psi(x_2) = \infty$.

```
1  int example3(int x1, int x2)
2    if (x1 > 42)
3      x1 += 2
4    if (x2 = 42)
5      x1 *= 2
6    return x1
```

To simplify the exposition, we limit the domain of x_1 and x_2 to $\mathcal{D} = D_8 = [-128, 127]$, assume the inputs to be distributed uniformally and let $x = (x_1, x_2)$ denote the tuple of variables. Hence the initial state is $I = \{x \in \mathcal{D} \times \mathcal{D}\}$ with an attacker's uncertainty of $H(\mathcal{U}) = \log 2^{8+8} = 16$. Applying algorithm Π we obtain the indistinguishability partition

$$\begin{aligned} I/_R = \{&\{x \in \mathcal{D} \times \mathcal{D} \mid x_1 > 42 \wedge x_2 = 42\}, \\ &\{x \in \mathcal{D} \times \mathcal{D} \mid x_1 > 42 \wedge x_2 \neq 42\}, \\ &\{x \in \mathcal{D} \times \mathcal{D} \mid x_1 \leq 42 \wedge x_2 = 42\}, \\ &\{x \in \mathcal{D} \times \mathcal{D} \mid x_1 \leq 42 \wedge x_2 \neq 42\}\}. \end{aligned}$$

The worst-case overall uncertainty after observation amounts $H_\infty(\mathcal{U}|\mathcal{V}_R) = 6.41$, so the maximum information flow of the program is $\mu_{max}(P) = 16 - 6.41 = 9.59$ bits. For checking the information flow policy Ψ we have to analyze the variables on its own. In the following we will focus on variable x_1, because x_2 will fulfill the policy in any case. Performing a projection on variable x_1 leads to a initial uncertainty of $H(\mathcal{U}_{|x_1}) = \log 2^8 = 8$ bits and a uncertainty after the observation of $H_\infty(U_{|x_1}|V_R) = \log 85 = 6.41$ bits. We obtain

$$\mu_{var}(P, x_1) = 8 - 6.41 = 1.59 \text{ bits.}$$

Since $\mu_{var}(P, x_1) \leq \Psi(x_1)$ and $\mu_{var}(P, x_2) \leq H(\mathcal{U}_{|x_2}) = 8 \leq \Psi(x_2)$, P complies to information flow policy Ψ.

Appendix B
Proof of Theorem 1

Proof. Given $A = (a_{i,j}) \in \mathbb{Z}^{n\times m}$, $b \in \mathbb{Z}^n$, $c \in \mathbb{Z}^m$ and $k \in \mathbb{Z}$, the *0-1 integer linear programming (0-1-ILP) decision problem* is to decide whether there exists an $x \in \{0,1\}^m$ such that $A \cdot x \leq b$ and $c^\top x \geq k$. This problem is known to be NP-complete. We perform a many-one reduction of the PASAPTO decision problem to the 0-1-ILP decision problem by transforming an 0-1-ILP instance I into a PASAPTO instance we denote by $\Gamma(I)$. Starting with the empty program P_0, we modify it as follows:

- For each column j of A we add a variable z_j and two control flow transitions t_j and $\tilde{t_j}$, which each leak 1 of z_j independently.
- For each row i of A we create a variable y_i. For all $a_{i,j} \in A$, if $a_{i,j} < 0$ then $\tilde{t_j}$ shall leak $-a_{i,j}$ of variable y_i, else t_j shall leak $a_{i,j}$ of variable y_i.
- Let $c_{sum} := \sum_{k=1}^{m} |c_k|$. For all c_j in c, if $c_j < 0$ then each branch of t_j needs $c_{sum} - c_j + 1$ time and each branch of $\tilde{t_j}$ needs $c_{sum}+1$ time, else each branch of t_j needs $c_{sum}+1$ and each branch of $\tilde{t_j}$ needs $c_{sum}+c_j+1$ time.
- Let $\delta : \mathbb{Z} \to \mathbb{N}$ with
$$\delta(z) := \begin{cases} 0, & \text{for } z \geq 0 \\ -z, & \text{for } z < 0 \end{cases}$$
For each row i of A we define the policy $\Psi(y_i) = b_i + \sum_{k=1}^{m} \delta(a_{i,k})$. For each column j of A define the policy $\Psi(z_j) = 1$.

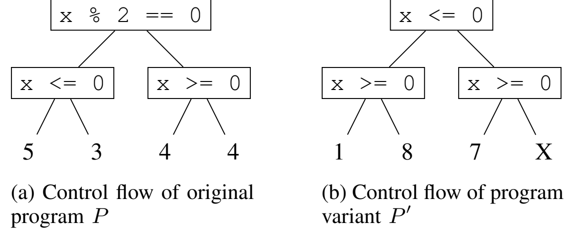
Let $k_p = 3m(c_{sum}+1)+2c_{sum}+\min(k, c_{sum})$ and $k_s = \infty$. For $x \in \{0,1\}^m$ we denote $\tilde{x}$ as vector with the same size as x and $\forall x_j : \tilde{x}_j := 1 - x_j$. We show that the output $\Gamma(I)$ equals the output of I.

Lemma 1. *For a 0-1-ILP I and its transformation $\Gamma(I) = (P, \Psi)$ it applies that:*

1) $\{x \mid \ A \cdot x \leq b\} = \left\{ x \mid \mathcal{T}\left(P, \begin{pmatrix} x \\ \tilde{x} \end{pmatrix}\right) \Psi\text{-compliant} \right\}$
We denote this set as S.
2) $\forall x \in S$:
$$c^\top \cdot x = \mu_p\left(\mathcal{T}\left(P, \begin{pmatrix} x \\ \tilde{x} \end{pmatrix}\right)\right) - 3m(c_{sum}+1) - 2c_{sum}$$
3) *Let*
$$F := \{(y,z)^\top \mid \mathcal{T}\left(P,(y,z)^\top\right) \Psi\text{-compliant} \wedge y \notin S\}$$
then $\forall x \in S \ \forall f \in F$:
$$\mu_p\left(\mathcal{T}\left(P, \begin{pmatrix} x \\ \tilde{x} \end{pmatrix}\right)\right) < \mu_p\left(\mathcal{T}\left(P, f\right)\right)$$
and
$$\mu_p\left(\mathcal{T}\left(P, f\right)\right) > 3m(c_{sum}+1) + 3c_{sum}$$

Proof. By construction. □

Lemma 1 shows that the PASAPTO decision problem is many-one reducible to the 0-1-ILP decision problem. This shows that the PASAPTO decision problem is NP-hard. □



(a) Control flow of original program P

(b) Control flow of program variant P'

Figure 10: Control flow graphs for proof in Appendix C

Appendix C
Proof of Theorem 2

Proof by Contradiction. Consider the following code of program P with input $x \in D_4$ distributed uniformly and the Maximum Information Flow measure μ_{max}.

```
1  int example4(int x)
2    if (x % 2 == 0)
3      if (x <= 0)
4        x += 1
5    else
6      if (x >= 0)
7        x -= 2
8    return x
```

The control flow of program P is illustrated in Figure 10a including the size of each indistinguishable equivalence class in the end of a program path. We obtain a worst-case information flow of $\mu_{max}(P) = 4 - \min\{\log 5, \log 3, \log 4\} = 2.42$ bits.

Consider the equivalent program $P' := \mathcal{T}(P, (0,1,1)^T)$. That means P' does not contain the first control flow transition (`x % 2 == 0`). To this end, it executes as well the then-branch as the else-branch of the hidden control flow transition. The adversary can now observe the control flow of both branches for every input. The control flow of P' is illustrated in Figure 10b, where 'X' denotes an infeasible path. The combination of branches leads to an increased information flow of $\mu_{max}(P') = 4 - \min\{\log 1, \log 8, \log 7\} = 4$ bits.

Summarizing,

$$(0,1,1)^T \leq_p (1,1,1)^T$$
$$\Rightarrow \mu_{max}(\mathcal{T}(P,(0,1,1)^T)) \geq \mu_{max}(\mathcal{T}(P,(1,1,1)^T)).$$

We can conclude that there is no monotonic increasing structure in general. □

Appendix D
Algorithms

We present the details of `GeneticPASAPTO` in Figure 11. Testing the dominance is implemented by the subroutine *dominates*(t_1, t_2) which on input two binary vectors outputs $\top$ if the program $\mathcal{T}(P, t_1)$ dominates the program $\mathcal{T}(P, t_2)$ and $\bot$ otherwise. Dominance is defined in terms of the objective function $f(t)$ given by the optimization problem. To simplify the depiction we use the function *updateFitness*(t_i, t_j), which reduces the fitness of individual j by 1 if *dominates*$(t_i, t_j) = \top$. By $J_{m,n}$ we denote the $m \times n$ matrix with all entries equal to one.

Require: P – Program under inspection
Ψ – Quantitative Information Flow Policy
N – Population size
α – Maximum Allowable Pareto Percentage
X – Maximum number of iterations
Ensure: $\mathbb{P}$ – Non-dominated set of compliant programs

1: **procedure** GeneticPASAPTO(P, Ψ, N, α, X)
2: $T := \mathit{filterControlFlow}(P)$
3: $V := \mathit{computeVariableSpace}(P)$
4: $R := 0_{|T|,N}$
5: /* Compute share parameter */
6: $\hat{P} := \mathcal{T}(P, 0_{|T|,1})$
7: $\sigma := \frac{2\mu_p(\hat{P}) - \mu_p(P)}{\mu_p(\hat{P})(N-1)}$
8: /* random start selection */
9: **for** $i := 1$ **to** N **do**
10: $R_{i,:} \leftarrow_{\$} \{0,1\}^{|T|}$
11: **end for**
12: /* initial fitness */
13: $F := J_{N,1}$
14: $F' := J_{N,1}$
15: **for** $i := 1$ **to** N **do**
16: $F_i := N$
17: **if** $\exists v \in V : \mu_{var}(\mathcal{T}(P, R_{:,i}), v) > \Psi(v)$ **then**
18: $F_i := 0$
19: **else**
20: **for** $j := 1$ **to** N **do**
21: $\mathit{updateFitness}(R_{:,j}, R_{:,i})$
22: **end for**
23: $F'_i := F'(i, R)$
24: **end if**
25: **end for**
26: $\mathit{paretoPercentage} := 0$
27: $\mathit{counter} := 1$
28: **while** $\mathit{paretoPercentage} \leq \alpha \land \mathit{counter} \leq X$ **do**
29: /* determine fittest and unfittest */
30: $\mathit{fittestInd} := \arg\max_{i \in N} F'_i$
31: $\mathit{fittest} := R_{:,\mathit{fittestInd}}$
32: $\mathit{secondFittestInd} := \max_{i \in N \setminus \{\mathit{fittestInd}\}} F'_i$
33: $\mathit{secondFittest} := R_{:,\mathit{secondFittestInd}}$
34: $a := \arg\min_{i \in N} F'_i$
35: $b := \arg\min_{i \in N \setminus \{a\}} F'_i$
36: /* crossing */
37: $\mathit{firstChild} := \mathit{fittest}$
38: $\mathit{secondChild} := \mathit{secondFittest}$
39: **for** $i := 1$ **to** $\lfloor \frac{|T|}{2} \rfloor$ **do**
40: $\mathit{firstChild}_i := \mathit{secondFittest}_i$
41: $\mathit{secondChild}_i := \mathit{fittest}_i$
42: **end for**
43: /* mutation */
44: $\mathit{randomVector} \leftarrow_{\$} \{n \in \mathbb{N} \mid 1 \leq n \leq N\}^{|T|}$
45: **for** $i := 1$ **to** $|T|$ **do**
46: **if** $\mathit{randomVector}_i = 1$ **then**
47: $\mathit{firstChild}_i := \mathit{firstChild}_i \oplus 1$
48: **end if**
49: **if** $\mathit{randomVector}_i = N$ **then**
50: $\mathit{secondChild}_i := \mathit{secondChild}_i \oplus 1$
51: **end if**
52: **end for**
53: **for** $i := 1$ **to** N **do**
54: /* update after removing old */
55: **if** $\mathit{dominates}(R_{:,a}, R_{:,i}) = \top$ **then**
56: $F_i := F_i + 1$
57: **end if**
58: **if** $\mathit{dominates}(R_{:,b}, R_{:,i}) = \top$ **then**
59: $F_i := F_i + 1$
60: **end if**
61: /* update after inserting new */
62: $\mathit{updateFitness}(\mathit{firstChild}, R_{:,i})$
63: $\mathit{updateFitness}(\mathit{secondChild}, R_{:,i})$
64: $F'_i := F'(i, R_{:,\cdot})$
65: **end for**
66: /* Reinitialize */
67: $R_{:,a} := \mathit{firstChild}$
68: $R_{:,b} := \mathit{secondChild}$
69: $F_a := N$
70: $F_b := N$
71: /* fitness of new */
72: **for** $i := 1$ **to** N **do**
73: $\mathit{updateFitness}(R_{:,i}, R_{:,a})$
74: $\mathit{updateFitness}(R_{:,i}, R_{:,b})$
75: **end for**
76: **if** $\exists v \in V : \mu_{var}(\mathcal{T}(P, R_{:,a}), v) > \Psi(v)$ **then**
77: $F_a := 0$
78: **end if**
79: **if** $\exists v \in V : \mu_{var}(\mathcal{T}(P, R_{:,b}), v) > \Psi(v)$ **then**
80: $F_b := 0$
81: **end if**
82: $F'_a := F'(a, R)$
83: $F'_b := F'(b, R)$
84: /* update loop */
85: $n := 0$
86: **for** $i := 1$ **to** N **do**
87: **if** $F_i = N$ **then**
88: $n := n + 1$
89: **end if**
90: **end for**
91: $\mathit{paretoPercentage} := \frac{n}{N}$
92: $\mathit{counter} := \mathit{counter} + 1$
93: **end while**
94: /* Prepare Return */
95: $\mathbb{P} := \emptyset$
96: **for** $i := 1$ **to** N **do**
97: **if** $F_i = \max_j F_j$ **then**
98: $\mathbb{P} := \mathbb{P} \cup \{\mathcal{T}(P, R_{:,i})\}$
99: **end if**
100: **end for**
101: **return** $\mathbb{P}$
102: **end procedure**

Figure 11: GeneticPASAPTO: Genetic Algorithm