Computation on Encrypted Data Using Dataflow Authentication

ANDREAS FISCHER, FZI Research Center for Information Technology, Germany
BENNY FUHRY, Camelot ITLab, Germany
JÖRN KUßMAUL and JONAS JANNECK, SAP Security Research, Germany
FLORIAN KERSCHBAUM, University of Waterloo, Canada
ERIC BODDEN, Heinz Nixdorf Institute, Paderborn University, and Fraunhofer IEM, Germany

Encrypting data before sending it to the cloud ensures data confidentiality but requires the cloud to compute on encrypted data. Trusted execution environments, such as Intel SGX enclaves, promise to provide a secure environment in which data can be decrypted and then processed. However, vulnerabilities in the executed program give attackers ample opportunities to execute arbitrary code inside the enclave. This code can modify the dataflow of the program and leak secrets via SGX side channels. Fully homomorphic encryption would be an alternative to compute on encrypted data without data leaks. However, due to its high computational complexity, its applicability to general-purpose computing remains limited. Researchers have made several proposals for transforming programs to perform encrypted computations on less powerful encryption schemes. Yet current approaches do not support programs making control-flow decisions based on encrypted data.

We introduce the concept of *dataflow authentication* (DFAuth) to enable such programs. DFAuth prevents an adversary from arbitrarily deviating from the dataflow of a program. Our technique hence offers protections against the side-channel attacks described previously. We implemented two flavors of DFAuth, a Java bytecode-to-bytecode compiler, and an SGX enclave running a small and program-independent trusted code base. We applied DFAuth to a neural network performing machine learning on sensitive medical data and a smart charging scheduler for electric vehicles. Our transformation yields a neural network with encrypted weights, which can be evaluated on encrypted inputs in 12.55 ms. Our protected scheduler is capable of updating the encrypted charging plan in approximately 1.06 seconds.

CCS Concepts: • **Security and privacy** → **Cryptography**; **Distributed systems security**; **Information flow control**; **Privacy protections**; *Privacy-preserving protocols*; • **Software and its engineering** → *Compilers;*

This work is an extended version of "Computation on Encrypted Data Using Dataflow Authentication" published in PoPETs 2020.1 [18].
The work of A. Fischer and B. Fuhry was primarily done at SAP Security Research.
This work was supported by the German Federal Ministry for Economic Affairs and Climate Action during the project "flexQgrid: Practical implementation of the quota-based grid traffic light concept for the utilization of flexibility from and within distribution grids" under grant number 03EI4002E.
Authors' addresses: A. Fischer, FZI Research Center for Information Technology, Germany; email: dfauth@andreasfischer.net; B. Fuhry, Camelot ITLab, Germany; J. Kußmaul and J. Janneck, SAP Security Research, Germany; F. Kerschbaum, University of Waterloo, Canada; email: fkerschbaum@uwaterloo.ca; E. Bodden, Heinz Nixdorf Institute, Paderborn University & Fraunhofer IEM, Germany; email: eric.bodden@upb.de.

2471-2566/2022/05-ART21 $15.00
https://doi.org/10.1145/3513005

Additional Key Words and Phrases: Trusted code base, trusted execution environment, homomorphic encryption, authenticated encryption, secure cloud computing

ACM Reference format:
Andreas Fischer, Benny Fuhry, Jörn Kußmaul, Jonas Janneck, Florian Kerschbaum, and Eric Bodden. 2022. Computation on Encrypted Data Using Dataflow Authentication. *ACM Trans. Priv. Secur.* 25, 3, Article 21 (May 2022), 36 pages.
https://doi.org/10.1145/3513005

1 INTRODUCTION

Many critical computations are being outsourced to the cloud. However, attackers might gain control of the cloud servers and steal the data they hold. End-to-end encryption is a viable security countermeasure but requires the cloud to perform computations on encrypted data.

Trusted execution environments such as Intel SGX enclaves [5, 28, 38] offer a potential solution to this problem. An SGX enclave can very efficiently run an entire program in encrypted memory, shielding it from the administrator's view. However, it already has been demonstrated that software vulnerabilities give attackers ample opportunities to execute arbitrary code in the enclave [35]. These attacks are called *return-oriented programming* and piece together programs from code snippets preceding return statements in the actual program. They can modify the control and data flow of the program and leak any secret in the program to an observer in the cloud via SGX side channels [13, 36, 45]. Since the number of software vulnerabilities grows with the size of the code base, it is advisable to keep the **trusted code base (TCB)** as small as possible. Hence, it is not recommended to outsource entire programs to an SGX enclave.

Consider the following dataflow modification attack that efficiently leaks a secret x in its entirety. Assume an encrypted variable $\mathsf{Enc}(x)$ in the domain $[0, N-1]$ is compared to $N/2-1$. The "then" branch is taken if it is lower or equal and the "else" branch otherwise. This can be observed, for example, by the branch shadowing attack presented by Lee et al. [36]. The observation of this behavior leaks whether $x \leq N/2-1$. This becomes quite problematic when assuming a strong, active adversary that can modify the control and data flow. The adversary may then create constants $\mathsf{Enc}(\bar{x})$ for $\bar{x} \in \{N/4, N/8, N/16, \ldots, 1\}$ in the program code, add those to the variable $\mathsf{Enc}(x)$, and re-run the control-flow branch. This way, by consecutively adding or subtracting the constants, the adversary can conduct a binary search for the encrypted value.

As a defense for this attack of modifying the dataflow, we introduce the concept of *dataflow authentication* (DFAuth). We instrument each control-flow decision variable with a label (broadly speaking, a **message authentication code (MAC)**) such that only variables with a preapproved dataflow can be used in the decision. Variables carry unique identifiers that are preserved and checked during the encrypted operations. This prevents an adversary from deviating from the dataflow in ways that would allow attacks such as the one mentioned earlier. Note that a program may still have *intentional* leaks introduced by the programmer. However, DFAuth restricts the leakage of any program to these intended leaks by the programmer that the programmer could avoid (e.g., by using appropriate algorithms such as data-oblivious ones). In essence, the technique restricts the information flows to those that are equivalent to the original program's information flows.

Fully homomorphic encryption (FHE) [23] would be another alternative to compute on encrypted data without the drawback of data leaks. Due to its high computational complexity [24], however, researchers are seeking efficient alternatives that offer similar security. Fortunately, we know how to efficiently perform additively and multiplicatively homomorphic operations on

Table 1. Comparison of DFAuth to the Most Relevant Alternative Approaches Computing on Encrypted Data

Approach	Support for Control Flow	Low Computational Overhead	Program-Independent TCB
FHE	○	○	●
SGX only	●	●	○
AutoCrypt	◐	●	●
DFAuth	●	●	●

encrypted data. Furthermore, if we reveal the control flow of a program (instead of computing a circuit), efficient computation seems feasible. Note that any control-flow decision on an encrypted variable is an intentional leak by the programmer. Several proposals for program transformation into such encrypted computations have been made. MrCrypt [49], JCrypt [15], and AutoCrypt [50] each offer an increasing set of programs that can be computed on encrypted data. To support encrypted computation on all programs, however, one needs to convert between different homomorphic encryption schemes. These conversions are very small routines such that we can scrutinize their code and implement them safely in a *trusted module* likely without any software vulnerabilities.

In this way, we combine the benefits of partially homomorphic encryption with a small TCB and the efficiency of unprotected program execution. Our re-encryption routines are small and program independent and are run protected in the trusted module, whereas the program runs efficiently on homomorphic encrypted values in unprotected memory. Hence, our approach significantly reduces the surface for attacks such as the return-oriented programming attacks described earlier. We take care not to destroy the benefits of outsourcing. The verification of labels is constant time and does not depend on the homomorphic computation. To this end, we introduce our own **homomorphic authenticated symmetric encryption (HASE)** scheme.

We complement DFAuth and HASE with an alternative concept for operating on ciphertexts. Our **trusted authenticated ciphertext operations (TACO)** scheme makes use of a common authenticated symmetric encryption scheme that does not support homomorphic evaluation on ciphertexts. As a result, ciphertext evaluations have to be performed in the trusted module, which hence needs to be invoked more often. However, our experiments show that the higher number of invocations is easily compensated by the use of a more efficient encryption scheme.

We implemented the program transformation in a bytecode-to-bytecode compiler such that the resulting programs are executable. We evaluated DFAuth based on two applications: a neural network performing machine learning on sensitive medical data and a smart charging scheduler for **electric vehicles (EVs)**. Our transformation yields a neural network with encrypted weights, which can be evaluated on encrypted inputs in 12.55 ms. Our protected scheduler is capable of updating the encrypted charging plan in approximately 1.06 seconds. This shows that DFAuth is practically deployable, while also providing extensive security guarantees.

For a summary of key properties provided by DFAuth and a comparison to the most relevant alternative approaches for computation on encrypted data, refer to Table 1. Note that FHE does not require any trusted code to be executed by the untrusted evaluator. Also note that AutoCrypt only supports control-flow decisions on encrypted *input* variables. DFAuth extends the state of the art to those programs performing control-flow decisions based on encrypted intermediate variables.

In summary, our contributions are the following:

- We define the concept of *dataflow authentication* (DFAuth) and show its interference equivalence property in a **program dependence graph (PDG)**.

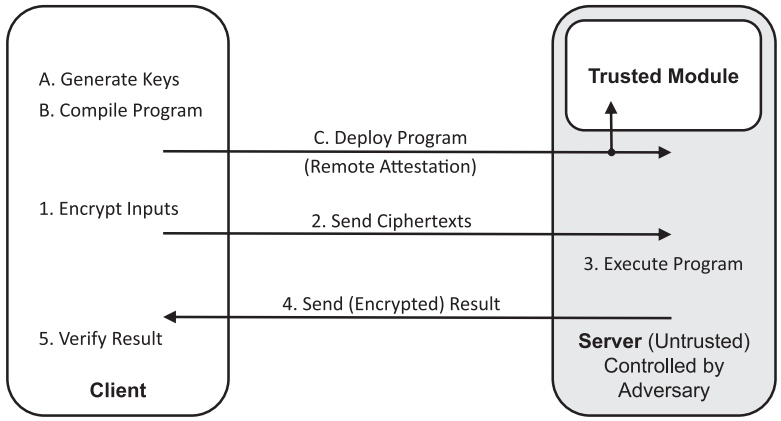

Fig. 1. System overview.

- We present two constant time implementations of dataflow authentication: *HASE* and *TACO*.
- We implemented and evaluated a *bytecode-to-bytecode program transformation* for computation on encrypted data using dataflow authentication.
- We implemented and evaluated transformed programs using an *Intel SGX* enclave as the trusted module.

Structure of this work. In Section 2, we provide our adversary model and define the syntax, correctness, and security of our HASE scheme. Based on HASE, we introduce DFAuth and the security it provides in Section 3. Section 4 presents our HASE constructions and discusses their security. We complement DFAuth and HASE with TACO in Section 5. Details about our implementation in Java are given in Section 6. Section 7 shows the results of our experiments using this implementation. Section 8 presents related work before Section 9 concludes our work.

2 DEFINITIONS

To understand the security of dataflow authentication, we first introduce the overall scenario, the adversary model, the algorithms that HASE offers, and the security it guarantees.

2.1 System Overview

We consider a scenario between a trusted client and an untrusted cloud server, which has a trusted (hardware) module (e.g., an Intel SGX enclave). The client wishes to execute a program at the cloud server with sensitive input data. An overview of the process and system trust boundaries are provided in Figure 1. We distinguish two phases of the outsourced computation: setup and runtime.

First, the client chooses the keys for the encryption of its inputs (A). Then, the client transforms the intended program using a specialized DFAuth-enabled compiler (B) and uploads it to the cloud. The server deploys some parts of the program into the trusted module that the client verifies by remote attestation (C). This concludes the setup phase.

In the runtime phase, the client can execute—multiple times if it wishes—the program on inputs of its choice. It encrypts the inputs using the information from the compiled program and sends the ciphertexts to the cloud server (1–2). The cloud server now executes the program (3). After the execution of the program the server returns an encrypted result to the client (4). The client can then verify the result of the computation (5).

2.2 Adversary Model

Our security objective is to leak only the information about the inputs to the cloud server that can be inferred from the program's executed control flow.

We assume an active adversary controlling the server who can do the following:

- *Read* the contents of all variables and the program text, except in the trusted module.
- *Modify* the contents of all variables and the program, except in the trusted module.
- *Continuously observe and modify* the control flow, such as by breaking the program, executing instructions step-by-step, and modifying the instruction pointer, except in the trusted module.
- Perform the preceding steps arbitrarily *interleaved*.

We require the following security goal: the server *learns nothing beyond the intended information flow* of the program to unclassified memory locations (interference equivalence as presented in Section 3).

Note that the remaining adversarial information flow can be minimized or eliminated by using appropriate algorithms such as data-oblivious ones or by combining DFAuth with control-flow obfuscation techniques. For example, code containing conditional instructions can be transformed into straight-line code [39] or both branches of a conditional can be executed and the result combined using an oblivious store operation [44]. We present a thorough analysis of the security and performance implications of control flow in related work [19].

2.3 Notation

We use the dot notation to access object members—for example, $O.A()$ refers to an invocation of algorithm A on object O. We use $:=$ for deterministic variable assignments and $=$ for comparisons. To indicate that an output of some algorithm may not be deterministic, we use $\leftarrow$ instead of $:=$ in assignments. We write $x \leftarrow\!\!\$\, X$ to sample x uniformly at random from a set X. For $m, n \in \mathbb{N}, m < n$, we use $[m, n]$ to refer to the set of integers $\{m, \ldots, n\}$. For a k-tuple $x = (x_1, x_2, \ldots, x_k)$, we refer to the projection of x onto its i-th ($i \in [1, k]$) component as $\pi_i(x) := x_i$. Similarly, for a set of k-tuples S, we define $\pi_i(S) := \{\pi_i(x) \mid x \in S\}$.

We follow the established convention of writing the group operation of an abstract group multiplicatively. Consequently, exponentiation refers to a repetition of the group operation. We may refer to a group $(\mathbb{G}, \cdot)$ simply as $\mathbb{G}$ if the group operation is clear from the context. With $s_1 \| s_2$, we denote the concatenation of bit strings s_1 and s_2.

Throughout the document, λ denotes a security parameter and 1^λ refers to the unary encoding of λ. The abbreviation *PPT* stands for *probabilistic polynomial time*. A function $f : \mathbb{N} \to \mathbb{R}^+$ is called *negligible* in n if for every positive polynomial p there is an n_0 such that for all $n > n_0$ it holds that $f(n) < 1/p(n)$. To indicate that some algorithm $\mathcal{A}$ is given black-box access to some function F, we write $\mathcal{A}^F$. Each parameter to F is either fixed to some variable or marked using a dot denoting that $\mathcal{A}$ may freely choose this parameter.

2.4 Game-Based Security

We provide security definitions as *games* (security experiments) played between a PPT *challenger* and a PPT *adversary* $\mathcal{A}$ [9]. The result of the game is 1 if $\mathcal{A}$ wins the game (i.e., breaks security) and 0 otherwise. $\mathcal{A}$'s *advantage* is defined as the probability of $\mathcal{A}$ winning the game minus the probability of trivially winning the game (e.g., by guessing blindly). Security holds if all adversaries have only a negligible advantage. The security proof is achieved by reducing the winning of the game to some problem that is assumed to be hard.

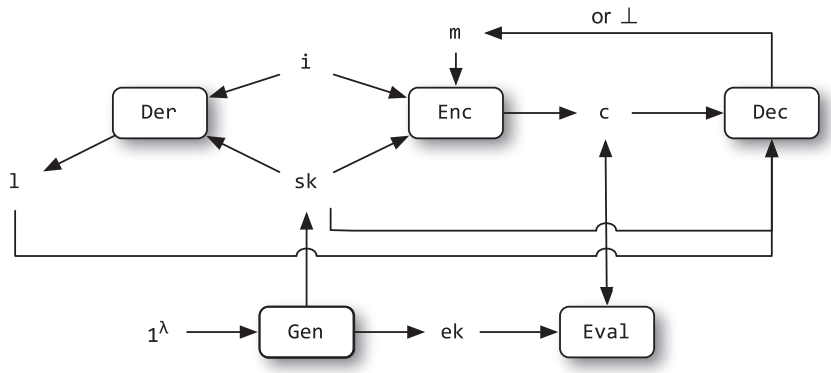

Fig. 2. HASE overview.

2.5 Homomorphic Authenticated Symmetric Encryption

In this section, we discuss the syntax, correctness, and security of a HASE scheme. For security, we define confidentiality in terms of indistinguishability and authenticity in terms of unforgeability. Indistinguishability of HASE schemes (HASE-IND-CPA) is defined as an adaptation of the commonly used IND-CPA security definition for symmetric encryption schemes [32, p. 74]. Unforgeability of HASE schemes (HASE-UF-CPA) is based on the common *unforgeable encryption* definition [32, p. 131].

Definition 1 (HASE Syntax). A HASE scheme is a tuple of PPT algorithms (Gen, Enc, Eval, Der, Dec) such that

- The *key-generation algorithm* Gen takes the security parameter 1^λ as input and outputs a key pair (ek, sk) consisting of a *public evaluation key* ek and a *secret key* sk. The evaluation key implicitly defines a commutative plaintext group $(\mathcal{M}, \oplus)$, a commutative ciphertext group $(C, \otimes)$, and a commutative label group $(\mathcal{L}, \diamond)$.
- The *encryption algorithm* Enc takes a secret key sk, a plaintext message $m \in \mathcal{M}$, and an identifier $i \in \mathcal{I}$ as input and outputs a ciphertext $c \in C$.
- The *evaluation algorithm* Eval takes an evaluation key ek and a set of ciphertexts $C \subseteq C$ as input and outputs a ciphertext $\hat{c} \in C$.
- The deterministic *label derivation algorithm* Der takes a secret key sk and a set of identifiers $I \subseteq \mathcal{I}$ as input and outputs a secret label $l \in \mathcal{L}$.
- The deterministic *decryption algorithm* Dec takes a secret key sk, a ciphertext $c \in C$ and a secret label $l \in \mathcal{L}$ as input and outputs a plaintext message $m \in \mathcal{M}$ or $\perp$ on decryption error.

An overview of all operations involved in our HASE scheme is provided in Figure 2.

Definition 2 (HASE Correctness). Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Der}, \mathsf{Dec})$ be a HASE scheme, $\mathcal{M}$ the plaintext group, and $\mathcal{I}$ the set of identifiers. We say that Π is *correct* if for all $(m_1, \ldots, m_n) \in \mathcal{M}^n$ with associated unique identifiers $(i_1, \ldots, i_n) \in \mathcal{I}^n$ there exists a negligible function $\mathsf{negl}(\lambda)$ such that

$$\Pr\left[\hat{m} = \bigoplus_{j \in [1,n]} m_j \;\middle|\; \begin{array}{l} (ek, sk) \leftarrow \mathsf{Gen}(1^\lambda) \\ \forall j \in [1,n] : c_j \leftarrow \mathsf{Enc}(sk, m_j, i_j) \\ l := \mathsf{Der}(sk, \{i_1, \ldots, i_n\}) \\ \hat{c} \leftarrow \mathsf{Eval}(ek, \{c_1, \ldots, c_n\}) \\ \hat{m} := \mathsf{Dec}(sk, \hat{c}, l) \end{array}\right] \geq 1 - \mathsf{negl}(\lambda)$$

where the probability is taken over the randomness of all algorithms.

$\underline{\text{ExpHASE}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(\lambda)}$
$(ek, sk) \leftarrow \Pi.\text{Gen}(1^\lambda)$
$(m_0, m_1, i_0, i_1, \text{st}) \leftarrow \mathcal{A}^{\Pi.\text{Enc}(sk,\cdot,\cdot)}(1^\lambda, ek)$
$b \leftarrow\!\!\$\ \{0,1\}$
$c \leftarrow \Pi.\text{Enc}(sk, m_b, i_b)$
$b' \leftarrow \mathcal{A}^{\Pi.\text{Enc}(sk,\cdot,\cdot)}(1^\lambda, c, \text{st})$
return $b = b'$

$\underline{\text{ExpHASE}^{\text{UF-CPA}}_{\mathcal{A},\Pi}(\lambda)}$
$S := \{\}$
$(ek, sk) \leftarrow \Pi.\text{Gen}(1^\lambda)$
$(c, I) \leftarrow \mathcal{A}^{E(sk,\cdot,\cdot)}(1^\lambda, ek)$
$l := \Pi.\text{Der}(sk, I)$
$m := \Pi.\text{Dec}(sk, c, l)$
if $I \not\subseteq \pi_2(S)$ **then**
  return $m \neq \bot$
else
  $\tilde{m} := \bigoplus_{(m',i)\in S, i\in I} m'$
  return $m \neq \bot \wedge m \neq \tilde{m}$

$\underline{E(sk, m, i)}$
if $i \in \pi_2(S)$ **then**
  return $\bot$
else
  $S := S \cup \{(m, i)\}$
  $c \leftarrow \Pi.\text{Enc}(sk, m, i)$
  return c

Fig. 3. HASE security experiments.

Definition 3 (HASE-IND-CPA). A HASE scheme Π has *indistinguishable encryptions under a chosen-plaintext attack*, or is *CPA secure*, if for all PPT adversaries $\mathcal{A}$ there is a negligible function $\text{negl}(\lambda)$ such that

$$\text{Adv}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(\lambda) := \left|\Pr\left[\text{ExpHASE}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(\lambda) = 1\right] - \tfrac{1}{2}\right| \leq \text{negl}(\lambda)$$

with the experiment defined as in Figure 3.

Definition 4 (HASE-UF-CPA). A HASE scheme Π is *unforgeable under a chosen-plaintext attack*, or just *unforgeable*, if for all PPT adversaries $\mathcal{A}$ there is a negligible function $\text{negl}(\lambda)$ such that

$$\text{Adv}^{\text{UF-CPA}}_{\mathcal{A},\Pi}(\lambda) := \Pr\left[\text{ExpHASE}^{\text{UF-CPA}}_{\mathcal{A},\Pi}(\lambda) = 1\right] \leq \text{negl}(\lambda)$$

with the experiment defined as in Figure 3.

The adversary returns a ciphertext c and a set of identifiers I. The adversary can be successful in two ways depending whether $I \subseteq \pi_2(S)$. If at least one $i \in I$ was not used in the encryption oracle, the adversary wins the game if c decrypts successfully under the label derived from I. If all $i \in I$ have been used in oracle queries, the adversary wins the game if c is correctly decrypted under the label derived from I and if the resulting plaintext m is different from the plaintext $\tilde{m}$ resulting from the application of the plaintext operation to the set of plaintexts corresponding to I. Note that by controlling I, the adversary controls which elements of S are used for the evaluation resulting in $\tilde{m}$.

3 DATAFLOW AUTHENTICATION (DFAUTH)

We introduce dataflow authentication (DFAuth) using the following example. Consider the excerpt from a program in Listing 1. First, DFAuth performs a conversion to **single static assignment (SSA)** form [4]: assign each variable at exactly one code location; create atomic expressions; introduce fresh variables if required. The resulting code is shown in Listing 2. As usual in SSA, `phi` is a specially interpreted merge function that combines the values of both assignments to `f`, here denoted by `f1` and `f2`. DFAuth then performs a type inference similar to JCrypt [15] and AutoCrypt [50]. As a result of this inference, each variable and constant is assigned an encryption type of $\{add, mul, cmp\}$. At runtime, each constant and variable value will be encrypted according to the

```
1  a = b + c;
2  d = a * e;
3  if (d > 42)
4   f = 1;
5  else
6   f = 0;
```

Listing 1. Example Code.

```
1  a = b + c;
2  d = a * e;
3  d1 = d > 42;
4  if (d1)
5   f1 = 1;
6  else
7   f2 = 0;
8  f = phi(f1,f2);
```

Listing 2. Example with SSA.

```
1  a = b + c;
2  a1 = convertToMul(a, "a1");
3  d = a1 * e;
4  d1 = convertToCmpGT42(d, "d1");
5  if (d1)
6   f1 = 1;
7  else
8   f2 = 0;
9  f = phi(f1,f2);
```

Listing 3. Example as Executed on the Server.

```
1  a = b + c;
2  a1 = convertToMul(a, "a1");
3  g1 = a1 * e;                  // changed
4  for(i = n..1) {               // inserted
5    g = phi(g1, g3);            // inserted
6    d = g + 2^i;                // inserted
7    d1 = convertToCmpGT42(d, "d1");
8    if (d1) {
9      f1 = 1;
10     g2 = g3 - 2^i;            // inserted
11   } else {
12     f2 = 0;
13     g3 = phi(g, g2);          // inserted
14     f = phi(f1,f2);
15     leak(f);                  // inserted
16   }
17 }
```

Listing 4. Example Modified by the Attacker.

appropriate type. HASE implements multiplicative homomorphic encryption *mul* and its operations directly, whereas it implements additive homomorphic encryption *add* using exponentiation. Comparisons *cmp* are implemented in the trusted module. Our experiments show that this is more efficient than performing the comparison in the program space using conversion to searchable or functional encryption. An attacker observing user space will hence only see encrypted variables and constants, but can observe the control flow. Actual data values are hidden from the attacker.

Combinations of multiple operations, however, require additional work. Every time a variable is encrypted in one encryption type (e.g., additive) but is later used in a different one (e.g., multiplicative), DFAuth must insert a conversion. The resulting program in our running example looks as follows.

The first conversion is necessary because the variable `a` must be converted from additive to multiplicative homomorphic encryption. The resulting re-encrypted value is stored in `a1`. For security reasons, the decryption performed by the conversion routine must be sensitive to the variable identifier it is assigned to. A unique label must be introduced to make the decryption routine aware of the code location. DFAuth can use the left-hand-side variable's identifier (`"a1"` in this example), because it introduced unique names during SSA conversion. Using this variable identifier, the conversion routine can retrieve the corresponding label of the HASE encryption stored in the memory protected by the trusted module.

Any branch condition is also treated as a conversion that leaks the result of the condition check. In the example, DFAuth introduces the variable `d1` to reflect this result:

```
4  d1 = convertToCmpGT42(d, "d1");
```

To simplify the exposition, we assume that our compiler inlines this comparison into a special routine `convertToCmpGT42`. In the general case, a binary comparison on two variables `x` and `y` would result in a call to a routine `convertToCmp(x,y,"z")`. We show the full algorithm in Listing 6 in Section 6, which is generic for all comparisons and in case of comparison to a constant looks this constant up in an internal table protected by the trusted module. We need to protect constants in comparisons because if they were part of the program text, they could be modified by the adversary.

As mentioned before, the security challenge of such conversions to *cmp* is that they leak information about the encrypted variables, and particularly that active adversaries that can modify the control and data flow can exploit those leaks to restore the variables' plaintext. In this article, we thus propose to restrict the dataflow using DFAuth. Secure conversions are enforced by allowing encrypted variables to only be decrypted along the program's original dataflow. The approach comprises two steps. First, happening at compile time, for each conversion DFAuth precomputes the Der algorithm (cf. Definition 1) on the operations in the code. In the conversion `convertToMul(a, "a1")` (at line 2 in our example), DFAuth computes the label

$$l2 = \mathsf{Der}(sk, \{\texttt{"b"}, \texttt{"c"}\})$$

and in the conversion at line 4

$$l4 = \mathsf{Der}(sk, \{\texttt{"a1"}, \texttt{"e"}\}).$$

Here the second argument to Der is the multi-set of variable identifiers involved in the unique computation preceding the conversion. We use a multi-set and not a vector because all of our encrypted operations are commutative. The compiler computes labels for all variables and constants in the program.

At runtime, the computed labels as well as the secret key *sk* are kept secret from the attacker, which is why both are securely transferred to, and stored in, the trusted module during the setup phase. The trusted module registers the secret labels under the respective identifier, for example, associating label $l4$ with identifier `"d1"`.

All conversion routines run within the trusted module. They retrieve a secret label for an identifier with the help of a `labelLookup(id)` function. In particular, when the program runs and a conversion routine is invoked, the trusted module looks up and uses the required labels for decryption. In the example at line 4, the call to `convertToCmpGT42` internally invokes the decryption operation `Dec(sk, d, l4)` using secret label `l4` retrieved for variable identifier `"d1"`:

```
1  convertToCmpGT42(d, "d1") {
2   l4 = labelLookup("d1");
3   x = Dec(sk, d, l4);
4   if (x == fail)
5     stop;
6   return (x > 42);
7  }
```

Note that in this scheme, the trusted module returns the result of the comparison in the clear. In this case, however, leaking the branch decision is *secure*, as HASE guarantees that any active attack that would yield the adversary a significant advantage will be reliably detected.

Let us assume an attacker that attempts to modify the program's data or control flow to leak information about the encrypted plaintexts, for instance, using a binary search as described in the introduction. The attacker is not restricted to the compiled instructions in the program and can also try to "guess" the result of cryptographic operations as the adversary in experiment $\mathrm{ExpHASE}^{\text{IND-CPA}}_{\mathcal{A},\Pi}$. This modification to the binary search algorithm can only succeed if the decryption operations Dec in `convertToCmpGT42` (or other conversion routines) succeed. The adversary can minimize the Dec operations, for example, by not introducing new calls to conversion routines, but given the scheme defined previously, any attempt to alter the dataflow on encrypted

variables will cause Dec to fail: assume that an attacker inserts code in Listing 3 to search for the secret value `d` (resulting code shown in Listing 4). We only use this code to illustrate potential attacks and ignore the fact that the attacker would need access to the encrypted constants (`2^i`) and therefore needs to guess the result of the homomorphic addition operation on the ciphertexts. However, given these capabilities, the attacker could try to observe the control flow—simulated by our statement `leak(f)`—which then would in turn leak the value of `d`.

This code will only execute if each variable decryption succeeds, but decryption for instance of `d1` will succeed only if it was encrypted with the same label `l4` that was associated with `d1` at load time. Since the trusted module keeps the labels and the key *sk* secret, the attacker cannot possibly forge the required label at runtime. Moreover, in the attacker-modified program, the encryption must fail due to the altered data dependencies: in the example, the input `d` to `convertToCmpGT42` has now been derived from `g3` and `i` instead of `a1` and `e`, which leads to a non-matching label for `d`. In result, the decryption in the conversion routine `convertToCmpGt42` will fail and stop program execution before any unintended leakage can occur.

General security argument. The way in which we derive labels from dataflow relationships enforces a notion of *interference equivalence*. A program P is said to be *non-interferent* [47] if applied to two different memory configurations M_1, M_2 that are equal w.r.t. their low (i.e., unclassified (unencrypted)) memory locations, $M_1 =_L M_2$ for short, then also the resulting memory locations after program execution must show such low-equivalency: $P(M_1) =_L P(M_2)$. Non-interference holds if and only if there is no information flow from high (i.e., classified (encrypted)) values to low memory locations. Although this is a semantic property, previous research has shown that one can decide non-interference also through a structural analysis of programs, through so-called PDGs that capture the program's control and data flow [51]. In this view, a program is *non-interferent* if the PDG is free of paths from high to low memory locations.

In the setting considered in this article, one must assume that the executed program *before encryption* already shows interference for some memory locations, for example, because the program is, in fact, intended to declassify some limited information (notably control-flow information). Let $M \downarrow C$ denote a projection of memory configuration M onto all (classified) memory locations C that are not declassified that way. Therefore, even in this setting, it holds for any program P and any memory configurations M_1, M_2 that $P(M_1 \downarrow C) =_L P(M_2 \downarrow C)$.

The main point of the construction proposed in this article is that any program that an attacker can produce, and that would lead to the same computation of labels (and hence decryptable data) as the original program, cannot produce any more information flows than the original program. Let us denote by tr a program transformation conducted by the attacker—for example, the transformation explained earlier, which inserted a binary search. Then the property we would like to obtain is that

$$\forall M_1, M_2, tr : P(M_1 \downarrow C) =_L P(M_2 \downarrow C) \Longrightarrow (tr(P))(M_1 \downarrow C) =_L (tr(P))(M_2 \downarrow C).$$

In other words, disregarding the explicitly declassified information within C, the transformed program does not leak any additional information—that is, the adversary cannot learn any additional information about the encrypted data. Let us assume for a moment that the preceding equation did not hold. If that were true, then there would exist a transformation tr that would cause the transformed program $tr(P)$ to compute values in at least one low memory location despite low-equivalent inputs. But this is impossible, as any such transformation would necessarily have to insert additional PDG-edges, destroying at least one label computation, and hence invalidating our HASE-UF-CPA security proof.

Result verification. Note that the client can verify the result of the computation using a simple check on the variable's label—just as the conversion routine does. The result is just another variable

that, as it is not converted, can be checked for correct dataflow computation. That way, a client can ensure that it receives a valid output of the program.

4 HASE CONSTRUCTIONS

In this section, we provide two constructions of HASE schemes: one homomorphic with respect to multiplication and another with respect to addition, on integers. We define the assumptions and show the security of our schemes under these assumptions. Security reductions are deferred to Appendix A.

Our first construction is based on the renowned public-key encryption scheme of Elgamal [17]. We do not make use of the public-key property of the scheme, but extend ciphertexts with a third group element working as a homomorphic authenticator.

Definition 5 (Group Generation Algorithm [32, p. 321]). A *group generation algorithm* is a PPT algorithm that takes 1^λ as input and outputs $(\mathbb{G}, q, g)$, where $\mathbb{G}$ is (a description of) a cyclic group, q is the order of $\mathbb{G}$, and g is a generator of $\mathbb{G}$.

Construction 1 (Multiplicative HASE). *Let $\mathcal{G}$ be a group generation algorithm. Define a HASE scheme using the following PPT algorithms:*

- Gen*: On input 1^λ obtain $(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^\lambda)$. For a* ***pseudorandom function (PRF)*** *family $H : \mathcal{K} \times \mathcal{I} \to \mathbb{G}$, choose $k \leftarrow_\$ \mathcal{K}$. Choose $a, x, y \leftarrow_\$ \mathbb{Z}_q$ and compute $h := g^x, j := g^y$. The evaluation key is $\mathbb{G}$, and the secret key is $(\mathbb{G}, q, g, a, x, y, h, j, k)$. The plaintext group is $(\mathcal{M}, \oplus) := (\mathbb{G}, \cdot)$, where $\cdot$ is the group operation in $\mathbb{G}$. The ciphertext group is $(\mathbb{G}^3, \otimes)$, where we define $\otimes$ to denote the component-wise application of $\cdot$ in $\mathbb{G}$. The label space is $(\mathbb{G}, \cdot)$.*
- Enc*: On input a secret key $sk = (\mathbb{G}, q, g, a, x, y, h, j, k)$, a message $m \in \mathbb{G}$, and an identifier $i \in \mathcal{I}$. Choose $r \leftarrow_\$ \mathbb{Z}_q$ and obtain the label $l = H(k, i)$. Compute $u := g^r, v := h^r \cdot m$ and $w := j^r \cdot m^a \cdot l$. Output the ciphertext (u, v, w).*
- Eval*: On input an evaluation key $\mathbb{G}$ and a set of ciphertexts $C \subseteq \mathcal{C}$ compute the ciphertext $c := \bigotimes_{c' \in C} c'$ and output c.*
- Der*: On input a secret key $(\mathbb{G}, q, g, a, x, y, h, j, k)$ and a set of identifiers $I \subseteq \mathcal{I}$ compute the label $l := \prod_{i \in I} H(k, i)$ and output l. Note that here Π denotes the repeated application of the group operation $\cdot$ in $\mathbb{G}$.*
- Dec*: On input a secret key $(\mathbb{G}, q, g, a, x, y, h, j, k)$, a ciphertext $c = (u, v, w)$, and a secret label $l \in \mathbb{G}$. First, compute $m := u^{-x} \cdot v$, then $t := u^y \cdot m^a \cdot l$. If t equals w, output m; otherwise, output $\perp$.*

It is well known that the Elgamal encryption scheme is homomorphic with regard to the group operation in $\mathbb{G}$. Trivially, this property is inherited by our construction. For the original Elgamal scheme, $\mathbb{G}$ is most commonly instantiated either as $\mathbb{G}_q$, the q-order subgroup of quadratic residues of $\mathbb{Z}_p^*$ for some prime $p = 2q + 1$ (with q also prime), or as an elliptic curve over some q-order finite field. In the latter case, the group operation is elliptic curve point addition and the ability to perform point addition in a homomorphism serves no useful purpose in our context. Instantiating $\mathbb{G}$ as $\mathbb{G}_q$ however enables homomorphic multiplication on the integers.

Our second construction supports homomorphic integer addition and is obtained by applying a technique proposed by Hu et al. [29] to Construction 1. The idea of this construction is to consider plaintexts to be element of $\mathbb{Z}_q$ instead of $\mathbb{G}$ and to encrypt a given plaintext m by first raising the generator g to the power of m and then encrypting the resulting group element in the usual way. In detail, this means computing ciphertexts of the form $(g^r, h^r g^m)$ rather than $(g^r, h^r m)$. To see that the resulting scheme is homomorphic with regard to addition on $\mathbb{Z}_q$, consider what happens

when the group operation is applied component-wise to two ciphertexts:

$$(g^{r_1} \cdot g^{r_2}, h^{r_1} g^{m_1} \cdot h^{r_2} g^{m_2}) = (g^{r_1+r_2}, h^{r_1+r_2} \cdot g^{m_1+m_2}).$$

Unfortunately, decryption now involves computing discrete logarithms with respect to base g, which must be difficult for sufficiently large exponents in order for the DDH problem (cf. Definition 7) to be hard relative to $\mathcal{G}$. Hu et al. [29] keep exponents small enough for discrete logarithm algorithms to terminate within reasonable time despite their exponential asymptotic running time. They do so by unambiguously decomposing plaintexts m into t smaller plaintexts m_e ($e \in [1, t]$) via means of the **Chinese remainder theorem (CRT)** and then encrypting each m_e separately. Although doing so increases the ciphertext size roughly by a factor of t in comparison to Construction 1, this drawback can be compensated by instantiating $\mathbb{G}$ as an elliptic curve group since the homomorphic operation is on $\mathbb{Z}_q$ rather than $\mathbb{G}$. At a comparable security level, group elements of elliptic curves can be represented using a fraction of bits [46].

We now provide the full details of our *Additive HASE* construction. Note how the authenticator only requires constant (i.e., independent of t) ciphertext space and can be verified without discrete logarithm computation. Although we consider instantiating $\mathbb{G}$ as an elliptic curve group, we keep writing the group operation multiplicatively.

Construction 2 (Additive HASE). *Let $\mathcal{G}$ be a group generation algorithm as before. Define a HASE scheme using the following PPT algorithms and the* Eval *algorithm from Construction 1:*

- Gen*: On input 1^λ obtain $(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^\lambda)$. For a PRF $H : \mathcal{K} \times \mathcal{I} \to \mathbb{Z}_q$, choose $k \leftarrow\!\!\$\, \mathcal{K}$. Choose $\{d_1, \ldots, d_t\} \subset \mathbb{Z}^+$ such that $d := \prod_{e=1}^{t} d_e < q$ and $\forall e \neq j : \gcd(d_e, d_j) = 1$. Define $D := (d_1, \ldots, d_t, d)$. Choose $a, x, y \leftarrow\!\!\$\, \mathbb{Z}_q$ and compute $h := g^x$, $j := g^y$. The evaluation key is $\mathbb{G}$, and the secret key is $(\mathbb{G}, q, g, a, x, y, h, j, k, D)$. The plaintext group is $(\mathcal{M}, \oplus) := (\mathbb{Z}_d, +)$. The ciphertext group is $(\mathbb{G}^{2(t+1)}, \otimes)$, where $\otimes$ denotes the component-wise application of $\cdot$ in $\mathbb{G}$. The label space is $(\mathbb{G}, \cdot)$.*
- Enc*: On input a secret key $sk = (\mathbb{G}, q, g, a, x, y, h, j, k, D)$, a message $m \in \mathbb{Z}_d$, and an identifier $i \in \mathcal{I}$. Obtain the label $l := H(k, i)$. For $e := 1, \ldots, t$:*
  - *Compute $m_e := m \bmod d_e$.*
  - *Choose $r_e \leftarrow\!\!\$\, \mathbb{Z}_q$.*
  - *Compute $u_e := g^{r_e}$.*
  - *Compute $v_e := h^{r_e} \cdot g^{m_e}$.*

  Choose $r \leftarrow\!\!\$\, \mathbb{Z}_q$. Compute $s := g^r$ and $w := j^r \cdot g^{m^a} \cdot l$. Output the ciphertext $(u_1, v_1, \ldots, u_t, v_t, s, w)$.
- Der*: On input a secret key $(\mathbb{G}, q, g, a, x, y, h, j, k, D)$ and a set of identifiers $I \subseteq \mathcal{I}$ compute the label $l := \prod_{i \in I} g^{H(k,i)}$ and output l.*
- Dec*: On input a secret key $(\mathbb{G}, q, g, a, x, y, h, j, k, D)$, a ciphertext $(u_1, v_1, \ldots, u_t, v_t, s, w)$, and a secret label $l \in \mathbb{G}$. Parse $D = (d_1, \ldots, d_t, d)$. First, compute $m_e := \log_g(v_e u_e^{-x})$ for $e = 1, \ldots, t$, then recover $m := \sum_{e=1}^{t} m_e \frac{d}{d_e} \left(\frac{d}{d_e}^{-1} \bmod d_e\right) \bmod d$. If $s^y \cdot g^{m^a} \cdot l = w$, then output m; else, output $\perp$. Note that $\log_g$ denotes the discrete logarithm with respect to base g.*

Definition 6 (Pseudorandom Function). Let X and Y be two finite sets and denote the set of all functions from X to Y as $\mathcal{F}$. We say that an efficiently computable keyed function $F : \mathcal{K} \times X \to Y$ with keyspace $\mathcal{K}$ is a PRF if for all PPT algorithms $\mathcal{A}$ there is a negligible function $\mathrm{negl}(\lambda)$ such that

$$\left|\Pr\left[\mathcal{A}^{F(k,\cdot)}(1^\lambda) = 1\right] - \Pr\left[\mathcal{A}^{f(\cdot)}(1^\lambda) = 1\right]\right| \leq \mathrm{negl}(\lambda)$$

where the first probability is taken over $k \leftarrow\!\!\$\, \mathcal{K}$ and the second probability is taken over $f \leftarrow\!\!\$\, \mathcal{F}$.

Definition 7 (DDH Problem [32, p. 321]). Let $\mathcal{G}$ be a group generation algorithm. We say that the **Decisional Diffie-Hellman (DDH)** *problem is hard relative to* $\mathcal{G}$ if for all PPT algorithms $\mathcal{A}$ there is a negligible function $\mathsf{negl}(\lambda)$ such that

$$\left|\Pr\left[\mathcal{A}(\mathbb{G}, q, g, g^{\alpha}, g^{\beta}, g^{\gamma}) = 1\right] - \Pr\left[\mathcal{A}(\mathbb{G}, q, g, g^{\alpha}, g^{\beta}, g^{\alpha\beta}) = 1\right]\right| \leq \mathsf{negl}(\lambda)$$

where in each case the probabilities are taken over the experiment in which $\mathcal{G}(1^{\lambda})$ outputs $(\mathbb{G}, q, g)$, and then $\alpha, \beta, \gamma \leftarrow\!\!\$\, \mathbb{Z}_q$.

Theorem 1 (Multiplicative HASE-IND-CPA). *Let* Π *be Construction 1. If the DDH problem is hard relative to* $\mathcal{G}$ *and* H *is a PRF as described in* Π.Gen*, then* Π *is CPA secure.*

Theorem 2 (Multiplicative HASE-UF-CPA). *Let* Π *be Construction 1. If* H *is a PRF as described in* Π.Gen*, then* Π *is unforgeable.*

Theorem 3 (Additive HASE-IND-CPA). *Let* Π *be Construction 2. If the DDH problem is hard relative to* $\mathcal{G}$ *and* H *is a PRF as described in* Π.Gen*, then* Π *is CPA secure.*

Theorem 4 (Additive HASE-UF-CPA). *Let* Π *be Construction 2. If* H *is a PRF as described in* Π.Gen*, then* Π *is unforgeable.*

5 TRUSTED AUTHENTICATED CIPHERTEXT OPERATIONS

In this section, we complement DFAuth and HASE with TACO, an alternative concept for operating on ciphertexts.

TACO is similar to HASE, but evaluations in TACO are defined as secret key operations, which offers some advantages. First, constructions do not have to rely on homomorphic encryption but can make use of more efficient symmetric encryption schemes. Second, the TACO syntax is more powerful and allows to perform multiplication, addition, and other operations such as division using the same generic construction. However, these properties are a trade-off: since the evaluation algorithm depends on the secret key, it must be run by a trusted party—that is, the client or the trusted module in our setting (cf. Section 2.2).

We define the syntax and correctness of a TACO scheme in Section 5.1. Section 5.2 provides our TACO security definitions. Section 5.3 presents a construction of a TACO scheme and its security properties. We again defer security reductions to Appendix A.

5.1 Syntax and Correctness

The syntax is similar to that of HASE. An important difference is that evaluation can be performed for multiple operations that do not necessarily have to be commutative. In addition, labels are public and their computation does not depend on the secret key.

Definition 8 (TACO Syntax). Let $\mathcal{M}$ be the message space, $\mathcal{C}$ the space of ciphertexts, $\mathcal{L}$ the space of labels, $\mathcal{K}$ the space of keys, and $\mathcal{I}$ the space of identifiers. Furthermore, let Φ be a set of plaintext operations. Each $\varphi \in \Phi$ has a fixed number of parameters p_{φ} such that φ maps a p_{φ}-dimensional tuple $(m_1, \ldots, m_{p_{\varphi}})$ with $m_j \in \mathcal{M}$ to one message $\hat{m} \in \mathcal{M}$. A TACO scheme is a tuple of PPT algorithms (Gen, Enc, Eval, Der, Dec) such that

- The *key-generation algorithm* Gen takes the security parameter 1^{λ} as input and outputs a secret key $sk \in \mathcal{K}$.
- The *encryption algorithm* Enc takes a secret key sk, a plaintext message $m \in \mathcal{M}$, and an identifier $i \in \mathcal{I}$ as input and outputs a ciphertext $c \in \mathcal{C}$.

- The *evaluation algorithm* Eval takes a secret key sk, a function $\varphi \in \Phi$, and a p_φ-dimensional tuple $(c_1, \ldots, c_{p_\varphi})$ with $c_j \in C$ as input and outputs a ciphertext $\hat{c} \in C$ or $\perp$ on authentication error.
- The deterministic *label derivation algorithm* Der takes either
  - an identifier $i \in \mathcal{I}$ or
  - a function $\varphi \in \Phi$ and a p_φ-dimensional tuple $(l_1, \ldots, l_{p_\varphi})$ with $l_j \in \mathcal{L}$

  as input and outputs a label $\hat{l} \in \mathcal{L}$.
- The deterministic *decryption algorithm* Dec takes a secret key sk, a ciphertext $c \in C$, and a label $l \in \mathcal{L}$ as input and outputs a plaintext message $m \in \mathcal{M}$ or $\perp$ on decryption error.

For a TACO scheme to be correct, the decryption must be successful and also yield the expected value even after multiple evaluations. For this purpose, we introduce the following definitions.

Definition 9 (TACO Partial Correctness). Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Der}, \mathsf{Dec})$ be a TACO scheme. We say that Π is *partially correct* if for all $m \in \mathcal{M}$ and all $i \in \mathcal{I}$ it holds that

$$\mathsf{Dec}(sk, \mathsf{Enc}(sk, m, i), \mathsf{Der}(i)) = m.$$

Definition 10 (TACO Ciphertext Validity). Let $c \in C$ be a ciphertext. We say c is *valid* if and only if $\exists l \in \mathcal{L} : \mathsf{Dec}(sk, c, l) \neq \perp$. The corresponding $l \in \mathcal{L}$, for which $\mathsf{Dec}(sk, c, l) \neq \perp$ is denoted as a *valid* label for c.

Definition 11 (TACO Correctness). Let $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Der}, \mathsf{Dec})$ be a partially correct TACO scheme. We say that Π is *correct* if for any secret key $sk \leftarrow \mathsf{Gen}(1^\lambda)$, any function $\varphi \in \Phi$, any p_φ-dimensional tuple of ciphertexts $C := (c_1, \ldots, c_{p_\varphi})$ with $c_j \in C$ and c_j *valid*, and any p_φ-dimensional tuple of labels $L := (l_1, \ldots, l_{p_\varphi})$ with $l_j \in \mathcal{L}$ and l_j *valid for* c_j, for the p_φ-dimensional tuple of plaintexts $M := (m_1, \ldots, m_{p_\varphi})$ with $m_j = \Pi.\mathsf{Dec}(sk, c_j, l_j)$ it holds that

$$\mathsf{Dec}(sk, \mathsf{Eval}(sk, \varphi, C), \mathsf{Der}(\varphi, L)) = \varphi(M).$$

5.2 Security Definitions

Indistinguishability of a TACO scheme is defined in a similar way to HASE-IND-CPA except the adversary is provided access to an evaluation oracle instead of the evaluation key. Like HASE-UF-CPA, unforgeability of a TACO scheme is based on the definition of unforgeable encryption [32, p. 131]. Essentially, the unforgeability adversary wins by producing two ciphertexts that decrypt to different messages under the same label.

Definition 12 (TACO-IND-CPA). A TACO scheme Π has *indistinguishable encryptions under a chosen-plaintext attack*, or is *CPA secure*, if for all PPT adversaries $\mathcal{A}$ there is a negligible function $\mathsf{negl}(\lambda)$ such that

$$\mathsf{Adv}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(\lambda) := \left| \Pr\left[\mathsf{ExpTACO}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(\lambda) = 1 \right] - \frac{1}{2} \right| \leq \mathsf{negl}(\lambda).$$

The experiment is defined as follows:

$$\begin{array}{l} \underline{\mathsf{ExpTACO}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(\lambda)} \\ sk \leftarrow \Pi.\mathsf{Gen}(1^\lambda) \\ (m_1, m_2, i_1, i_2, \mathrm{st}) \leftarrow \mathcal{A}^{\Pi.\mathsf{Enc}(sk,\cdot,\cdot),\Pi.\mathsf{Eval}(sk,\cdot,\cdot)}(1^\lambda) \\ b \leftarrow\!\!\$\, \{0,1\} \\ c \leftarrow \Pi.\mathsf{Enc}(sk, m_b, i_b) \\ b' \leftarrow \mathcal{A}^{\Pi.\mathsf{Enc}(sk,\cdot,\cdot),\Pi.\mathsf{Eval}(sk,\cdot,\cdot)}(1^\lambda, c, \mathrm{st}) \\ \textbf{return } b = b' \end{array}$$

Definition 13 (TACO-UF-CPA). A TACO scheme Π is *unforgeable under a chosen-plaintext attack*, or just *unforgeable*, if for all PPT adversaries $\mathcal{A}$ and all functions $\varphi \in \Phi_\Pi$ there is a negligible function $\mathsf{negl}(\lambda)$ such that

$$\mathrm{Adv}^{\text{UF-CPA}}_{\mathcal{A},\Pi}(\lambda) := \Pr\left[\mathrm{ExpTACO}^{\text{UF-CPA}}_{\mathcal{A},\Pi,\varphi}(\lambda) = 1\right] \leq \mathsf{negl}(\lambda)$$

with the experiment defined as follows:

$\mathrm{ExpTACO}^{\text{UF-CPA}}_{\mathcal{A},\Pi,\varphi}(\lambda)$

$I := \{\}$

$sk \leftarrow \Pi.\mathsf{Gen}(1^\lambda)$

$(c_1, c_2, l) \leftarrow \mathcal{A}^{E(sk,\cdot,\cdot),\Pi.\mathsf{Eval}(sk,\cdot,\cdot)}(1^\lambda)$

$m_1 := \Pi.\mathsf{Dec}(sk, c_1, l)$

$m_2 := \Pi.\mathsf{Dec}(sk, c_2, l)$

return $m_1 \neq \bot \wedge m_2 \neq \bot \wedge m_1 \neq m_2$

$E(sk, m, i)$

if $i \in I$ **then**

  return $\bot$

else

  $I := I \cup \{i\}$

  $c \leftarrow \Pi.\mathsf{Enc}(sk, m, i)$

  return c

5.3 Construction

Definition 14 (Symmetric Encryption Scheme [32, p. 6]). A *symmetric encryption scheme* for key space $\mathcal{K}$, message space $\mathcal{M}$. and ciphertext space C is a triple of PPT algorithms (Gen, Enc, Dec) such that

- Gen takes as input the security parameter 1^λ and outputs a key $k \in \mathcal{K}$.
- Enc takes as input a key $k \in \mathcal{K}$ and a plaintext message $m \in \mathcal{M}$ and outputs a ciphertext $c \in C$.
- Dec takes as input a key $k \in \mathcal{K}$ and a ciphertext $c \in C$ and outputs a message $m \in \mathcal{M}$.

Correctness requires for any key $k \leftarrow \mathsf{Gen}(1^\lambda)$ and every $m \in \mathcal{M}$ to hold: $\mathsf{Dec}(k, \mathsf{Enc}(k, m)) = m$.

Definition 15 (Hash Function [32, p. 154]). A *hash function* (with output length l) is a pair of PPT algorithms (Gen, H) such that

- Gen takes as input the security parameter 1^λ and outputs a key s.
- H takes as input a key s and a string $x \in \{0,1\}^*$ and outputs a string $y \in \{0,1\}^{l(\lambda)}$, where λ corresponds to the security parameter used by Gen to generate s.

Construction 3 (TACO). *Define the message space $\mathcal{M} := \{0,1\}^*$, the ciphertext space $C := \{0,1\}^*$, the label space $\mathcal{L} := \{0,1\}^\lambda$, and the space of identifiers $\mathcal{I} := \{0,1\}^*$. Let $SE = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ be a symmetric encryption scheme with message space $\mathcal{M}$ and ciphertext space C. Define the key space $\mathcal{K}$ as the space of keys output by $SE.\mathsf{Gen}$. For a hash function $\Omega = (\mathsf{Gen}, H)$, obtain $k \leftarrow \Omega.\mathsf{Gen}(1^\lambda)$. Furthermore, let Φ be a set of plaintext operations and let $\mathtt{id} : \Phi \mapsto \{0,1\}^*$ be an injective function. Construct a TACO scheme using the following PPT algorithms:*

- Gen*: On input 1^λ obtain and output a symmetric encryption secret key $sk \leftarrow SE.\mathsf{Gen}(1^\lambda)$.*
- Enc*: On input a secret key sk and a message $m \in \mathcal{M}$ output the ciphertext*

$$c \leftarrow SE.\mathsf{Enc}(sk, m \| \mathsf{Der}(i)).$$

- Eval*: On input a secret key sk, an operation $\varphi \in \Phi$, and a p_φ-dimensional tuple of ciphertexts $(c_1, \ldots, c_{p_\varphi})$. For $j \in [1, p_\varphi]$, compute $d_j := SE.\mathsf{Dec}(sk, c_j)$ and parse $d_j = m_j \| l_j$. If any $d_j = \bot$, then return $\bot$. Otherwise, compute*

$$m' := \varphi\left((m_1, \ldots, m_{p_\varphi})\right) \text{ and } l' := \mathsf{Der}\left(\varphi, (l_1, \ldots, l_{p_\varphi})\right).$$

Output the ciphertext $\hat{c} \leftarrow SE.\mathsf{Enc}\,(sk, m' \| l')$.

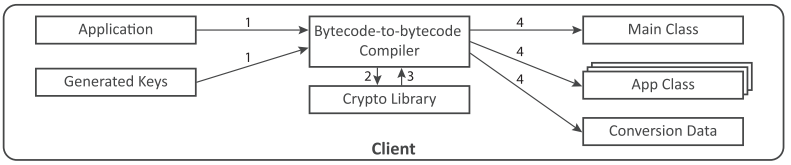

Fig. 4. Application transformation during the setup phase. See Section 6.1 for a full description.

- Der*:*
  - *On input a secret key sk and an identifier* $i \in \mathcal{I}$ *output the label*
$$\hat{l} := H(k, i\|0).$$
  - *On input a secret key sk, a function* $\varphi \in \Phi$*, and a* p_φ*-dimensional tuple of labels* $(l_1, \ldots, l_{p_\varphi})$ *output the label*
$$\tilde{l} := H\left(k, \mathtt{id}(\varphi)\|l_1\| \ldots \|l_{p_\varphi}\|1\right).$$
- Dec*: On input a secret key sk, a ciphertext* $c \in \mathcal{C}$*, and label* $l \in \mathcal{L}$ *compute* $d := SE.Dec(sk, c)$ *and parse* $d = m'\|l'$*. If* $d = \perp$ *or* $l \neq l'$*, then output* $\perp$*. Otherwise, output* m'*.*

Theorem 5 (TACO-IND-CPA). *Let* Π *be Construction 3 and SE its symmetric encryption scheme. If SE is CCA-secure [32, p. 96], then* Π *is* TACO-IND-CPA *secure.*

Theorem 6 (TACO-UF-CPA). *Let* Π *be Construction 3, SE its symmetric encryption scheme, and* $\Omega = (\mathrm{Gen}, H)$ *its hash function. If SE is unforgeable [32, p. 131] and* Ω *is collision resistant [32, p. 155], then* Π *is* TACO-UF-CPA *secure.*

6 IMPLEMENTATION

In this section, we present details of an implementation in Java used in our experiments. Recall from Section 2.1 that we consider a scenario between a trusted *client* and an untrusted cloud *server* that has a *trusted module*. Also recall that we distinguish two phases of the outsourced computation: *setup* and *runtime*.

6.1 Setup Phase

The setup phase is divided into two parts, *compilation* and *deployment*, as described in the following. An overview is provided in Figure 4.

Compilation. First, the client translates any Java bytecode program to a bytecode program running on encrypted data. To start, the client generates a set of cryptographic keys. It then uses our bytecode-to-bytecode compiler to transform an application (in the form of Java bytecode) using the generated keys (1). Our compiler is based on Soot, a framework for analyzing and transforming Java applications [34]. It uses our DFAuth crypto library to encrypt program constants and choose variable labels (2–3).

The DFAuth crypto library contains implementations of all required cryptographic algorithms, including our own from Sections 4 and 5. It implements the PRF used for authentication labels as HMAC-SHA256 [16]. For the group operations in Multiplicative HASE, we use MPIR [1] for large integer arithmetic. Additive HASE operates on the elliptic curve group provided by `libsodium` [3]. The Gen method of Additive HASE has as parameters the number of ciphertext components and the number of bits per component. From these, it deterministically derives a set of t primes. The Additive HASE Dec method computes the discrete logarithms via exhaustive search with a fixed

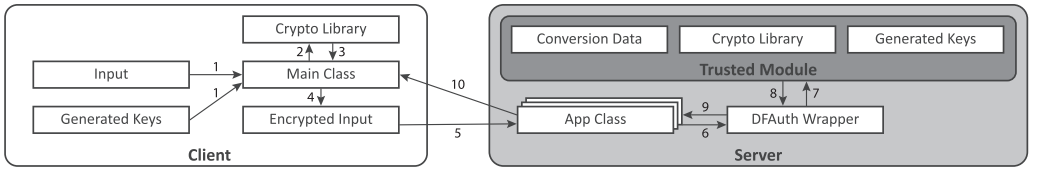

Fig. 5. Application execution during the runtime phase. See Section 6.2 for a full description.

set of precomputed values. To ensure the efficiency of Additive HASE decryption, the compiler inserts trusted module invocations into the program that decrypt and re-encrypt ciphertexts. These re-encryptions result in modulo reductions in the exponents (cf. Construction 2), thus preventing excessive exponent growth and ensuring an efficient decryption. The frequency of these invocations can be defined by the application. We demonstrate the efficacy of these re-encryptions in Appendix B. For HASE, our compiler converts floating-point plaintexts to a fixed-point representation using an application-defined scaling factor. It also transforms the calculations to integer values, whereby the scaling factors are considered when appropriate. The resulting value is transformed back to floating-point after decryption. For the symmetric encryption scheme in TACO we use the Advanced Encryption Standard in Galois/Counter Mode (AES-GCM). AES-GCM is an authenticated encryption scheme and as such is both CCA secure and unforgeable.

Finally, the compiler performs the transformation described in Section 3 and outputs a *main class* containing the program start code, multiple *app classes* containing the remaining code, and *conversion data* (e.g., labels and comparison data) (4).

Deployment. Second, the client deploys the app classes at the cloud server and securely loads the generated cryptographic keys and conversion data into the trusted module. We implemented the trusted module using an Intel SGX enclave [5, 28, 38]. SGX is well suited for our implementation because it provides isolated program execution (including strong memory encryption) and remote attestation (including a secure communication channel). The client uses remote attestation to prepare the enclave. It verifies the correct creation of the DFAuth trusted module enclave in the remote system and the correct setup of the crypto library. At the same time, the client establishes a secure communication channel with the remote enclave, over which the sensitive conversion data is loaded. The secure channel provides confidentiality and authenticity. It protects the communication between the trusted client and the trusted enclave against the untrusted part of the server as well as any other attackers on the network. We also emphasize that SGX's hardware protections protect cryptographic keys and conversion data on the server from access by any software except our DFAuth enclave.

6.2 Runtime Phase

To run the program, the client executes the main class that triggers the remote program execution at the cloud server (Figure 5). The main class encrypts the *program input* (for this run of the program) with the generated keys (for the entire setup of the program) using the crypto library (1–4). The main class passes the *encrypted input* to the app classes on the cloud server (5). The app classes operate on encrypted data and do not have any additional protection. They invoke the *DFAuth wrapper* for operations on homomorphic ciphertexts and re-encryption or comparison requests (6). The wrapper hides the specific homomorphic encryption schemes and trusted module implementation details from the app classes such that it is feasible to run the same program using different encryption schemes or trusted modules. It forwards re-encryption and comparison requests to the trusted module and passes the answers back to the application (7–9). Once the

```
convertToMul(x, "x") {
  label = labelLookup("x");
  y = Dec(K, x, label);
  if (y == fail)
    stop;
  id = idLookup("x");
  return Enc(K, y, id);
}
```

Listing 5. Conversion to Multiplicative HASE.

```
convertToCmp(x, y, "x") {
  label = labelLookup("x");
  x1 = Dec(K, x, label);
  if (x1 == fail)
    stop;
  if (y == null) {
    param = paramLookup("x");
    switch (param.type) {
      case EQ:
        return (x1 == param.const);
      case GT:
        return (x1 > param.const);
      case GTE:
        ...
  } else {
    label = labelLookup("y");
    y1 = Dec(K, y, label);
    if (y1 == fail)
      stop;
    ...
  }
}
```

Listing 6. Conversion to Comparison.

app classes finish their computation, they send an encrypted result (including an authentication label) back to the client (10). The client verifies the authentication label to the one computed by our compiler.

The task of the trusted module during runtime is to receive re-encryption and comparison requests, determine whether they are legitimate, and answer them if they are. It bundles cryptographic keys, authentication labels, and required parts of the crypto library inside a trusted area, shielding it from unauthorized access. The DFAuth wrapper enables to potentially select different trusted modules based on the client's requirements and their availability at the cloud server. Besides Intel SGX enclaves, one can implement a trusted module using a hypervisor or calling back to the client for sensitive operations. However, alternative implementations would involve making slightly different trust assumptions.

SGX's secure random number generator provides the randomness required during encryption. A restriction of the current generation of Intel SGX is the limited size of its isolated memory. It only provides about 96 MB for code and data and thus enforces an upper bound on the number of precomputed discrete logarithm values used to speedup Additive HASE. The available memory can be used optimally with a careful selection of CRT parameters.

TACO evaluation, HASE re-encryption, and comparison requests have to be implemented inside the trusted module. We display the conversion routines (implemented in an SGX enclave in our case) for conversion to Multiplicative HASE and comparison in Listings 5 and 6. The conversion routine to Additive HASE is similar to the one for Multiplicative HASE in Listing 5 with the roles of the encryption schemes switched. The comparison of two encrypted values is similar to the comparison of one to a constant in Listing 6. Similar to the call `labelLookup`, which retrieves labels from conversion data stored inside the trusted module, `idLookup` and `paramLookup` retrieve identifiers for encryption and parameters for comparison from the conversion data.

7 EVALUATION

In this section, we present the evaluation results collected in two experiments. In the first experiment, we apply DFAuth to an existing neural network program enabling secure neural network

evaluation in the cloud. In the second experiment, we use DFAuth to protect sensitive data processed by a smart charging scheduler for EVs. In each experiment, we separately evaluate DFAuth with HASE and DFAuth with TACO. We collect the running time inside and outside the trusted module as well as the number of operations performed inside and outside the trusted module.

All experiments were performed in the Microsoft Azure Cloud using Azure Confidential Computing. We used VM instances of type `Standard_DC4s`, which run Ubuntu Linux 18.04 and have access to four cores of an SGX-capable Intel Xeon E-2176G CPU and 16 GiB of RAM. We aimed for a security-level equivalent to 80 bits of symmetric encryption security. We used the 1,536-*bit MODP Group* from RFC3526 [33] as the underlying group in Multiplicative HASE. The `libsodium` [3] elliptic curve group used by Additive HASE even provides a security level of 128 bits [10]. A key length of 128 bits was used for the AES-GCM symmetric encryption scheme in TACO.

7.1 Secure Neural Networks in the Cloud

In this experiment, we consider the use case of evaluating neural networks in the cloud. Due to their computational complexity, it is desirable to outsource neural network computations to powerful computing resources located at a cloud service provider.

We aim to protect the network model and the instance classified—that is, the weights of the connections and the inputs and outputs of the neurons. The weights do not change between classifications and often represent intellectual property of the client. In addition, the privacy of a user classified by the network is at risk, since his classification may be revealed. DFAuth overcomes these concerns by encrypting the weights in the network and the client's input and performing only encrypted calculations. As well, since the transformed program does not perform any control-flow decisions based on network weights or client input, the attacker cannot learn sensitive data by observing the control flow. Note that even the classification result does not leak, since the result returned is the output values for each of the classification neurons (i.e., a chance of classification y, e.g., breast cancer in our subsequent example, of $x\%$).

Experimental setup. We apply our transformation to the BrestCancerSample [sic] neural network provided by Neuroph [2]. This network is a fully connected multi-layer perceptron with 30 input neurons, 16 hidden neurons, and two output neurons. It uses the sigmoid function as its activation function. Given a set of features extracted from an image of a breast tumor, the network predicts whether the tumor is malignant or benign. As such, it operates on highly sensitive medical data. The properties of the network (e.g., layer and neuron configuration) are encoded programmatically in the main class of this network. This class also reads the dataset associated with the network and divides it into a 70% training set and a 30% test set. The training set is used to learn the network, and the test set is used to evaluate whether the network delivers correct predictions.

We start by applying DFAuth to the main class of the network and the classes of the framework (app classes). Result of the transformation is a new main class and a set of app classes operating on ciphertexts rather than floating-point double values. Floating-point numbers are converted to fixed-point numbers by scaling by a factor of 10^6. We use the facilities provided by Neuroph to serialize the trained network weights into a double array and encrypt each weight. The encrypted weights and the network configuration form the encrypted neural network. We use Neuroph to write the encrypted neural network to disk just like the original one operating on plaintext.

For both the plaintext and encrypted neural network, we test different network evaluation sizes ($\{1, 10, 20, \ldots, 100\}$) and perform 20 runs each. For every run, a new random segmentation of training and test data is performed and the network is trained again. Network inputs are sampled uniformly at random (without replacement) from the test dataset. We measure the total running time of code executing at the untrusted server, the time spent invoking and inside the trusted

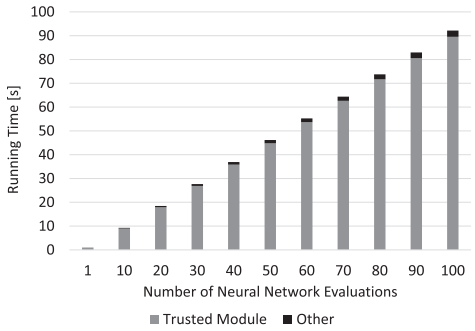

Fig. 6. Mean running time (in seconds) of the HASE variant of the breast cancer neural network experiment as a function of the number of evaluations.

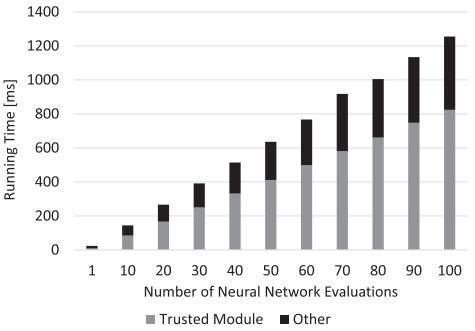

Fig. 7. Mean running time (in milliseconds) of the TACO variant of the breast cancer neural network experiment as a function of the number of evaluations.

module, and the number of operations performed on encrypted data inside and outside of the trusted module. The total running time includes reading the network configuration (i.e., layers and neuron), loading the weights, and executing the evaluation.

Evaluation results. Figure 6 presents the mean running time of the encrypted neural network using DFAuth with HASE. The mean is computed over all 20 runs for each evaluation size and is divided into time spent inside the trusted module and time spent outside the trusted module. Figure 7 shows the results for DFAuth with TACO. Figure 8 compares the running times of HASE and TACO. We do not include the plaintext measurements in the graphs because they are too small to be visible. Table 2 reports the number of untrusted and trusted module (SGX) operations.

Using HASE, the total running time of one network evaluation is 951 ms, whereby 895 ms (94.1%) are spent in the trusted module (SGX) and 56 ms (5.9%) outside of the trusted module. Even for 100 evaluations, a run completes in 92.15 seconds on average. In this case, the processing time in the trusted module is 89.54 seconds (97.2%) and 2.60 second (2.8%) outside. The relative running time of an evaluation (total running time / number of network evaluations) is 921 ms. Compared to one plaintext network evaluation, the running time increased by a factor of about 1,417. A waiting time of less than 1 second demonstrates the practical deployment of neural network evaluation on encrypted data and should already be acceptable for most use cases.

In Figure 6, we can see that using HASE a significant portion of the total runtime is spent inside (or invoking) the trusted module. On the one hand, this shows that a more efficient trusted module implementation would significantly decrease the total runtime of the application. On the other hand, it suggests that we execute more instructions inside the trusted module than outside, contradicting our basic idea of a reduced execution inside the trusted module. However, Table 2 shows that this is not the case. For a single neural network evaluation, 1,096 untrusted operations and 620 trusted operations on encrypted data are performed. This means that 64% of all operations can be performed without the trusted module.

It is important to note that for each run and every input, the prediction of the encrypted network was consistent with the prediction of the plaintext network—that is, DFAuth introduced no additional error due to the encrypted computation.

Using TACO, the total running time of one network evaluation is 23 ms, of which 8 ms (35.7%) are spent in the trusted module (SGX) and 15 ms (64.3%) outside of the trusted module. For 100 evaluations, the relative running time of an evaluation is 12.55 ms. In Figure 7, we can see that in

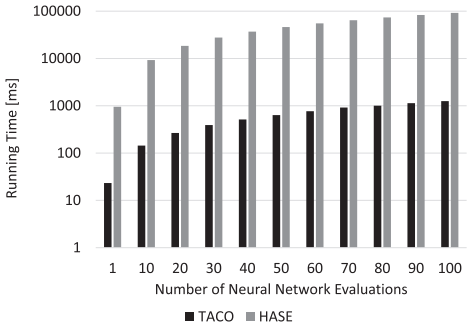

Fig. 8. Compared mean running time (in milliseconds) of the TACO and the HASE variant of the breast cancer neural network experiment as a function of the number of evaluations.

Table 2. Number of Untrusted Homomorphic (HOM) and Trusted (SGX) Operations for a Single Evaluation of the Neural Network

Operation	HASE		TACO	
	Type	**# Ops**	**Type**	**# Ops**
Addition	HOM	548	SGX	548
Multiplication	HOM	548	SGX	548
AddToMul Conversion	SGX	36		
MulToAdd Conversion	SGX	548		
Comparison	SGX	36	SGX	36
Total	HOM	1096	SGX	1132
	SGX	620		

contrast to HASE, a smaller fraction of the total running time is spent in the trusted module. Although TACO requires 1,132 trusted module invocations while HASE only requires 620, it appears that the higher number of invocations is easily compensated by TACO's use of a more efficient encryption scheme. For one network evaluation, TACO is faster than HASE by a factor of 41 and slower than plaintext evaluation by a factor of 35. See Figure 8 for a more detailed comparison of HASE and TACO running times.

Comparison to alternative solutions. Recently, implementations of machine learning on encrypted data have been presented for somewhat homomorphic encryption [25] and Intel SGX [41]. Compared to the implementation on somewhat homomorphic encryption, our approach offers the following advantages. First, our approach has a *latency* of 12.55 ms compared to 570 seconds for somewhat homomorphic encryption. The implementation in the work of Gilad-Bachrach et al. [25] exploits the inherent parallelism of somewhat homomorphic encryption to achieve a high throughput. However, when evaluating only one sample on the neural network, the latency is large. Our approach is capable of evaluating only a single sample with low latency as well. Second, our approach scales to different machine learning techniques with *minimal developer effort*. Whereas the algorithms in the work of Gilad-Bachrach et al. [25] were developed for a specific type of neural network, our implementation on encrypted data was derived from an existing implementation of neural networks on plaintext data by compilation. This also implies that the error introduced by Gilad-Bachrach et al. [25] due to computation on integers does not apply in our case. However, we have not evaluated this aspect of accuracy in comparison to the work of Gilad-Bachrach et al. [25]. Finally, our approach is capable of *outsourcing* a neural network evaluation, whereas the approach in the work of Gilad-Bachrach et al. [25] is a two-party protocol (i.e., the weights of the neural network are known to the server). Our approach encrypts the weights of the neural network and hence a client can outsource the computation of neural network. Note that our approach includes the functionality of evaluating on plaintext weights as well and hence offers the larger functionality.

Although their running time overhead is smaller than ours, our approach offers the following advantage compared to the implementation on SGX [41]. In our approach, the code in the SGX enclave is *independent of the functionality* (e.g., machine learning). The implementation in the work of Ohrimenko et al. [41] provides a new, specific algorithm for each additional machine learning function (neural networks, decision trees, etc.). Each implementation has been specifically optimized to avoid side channels on SGX and hopefully scrutinized for software vulnerabilities. The

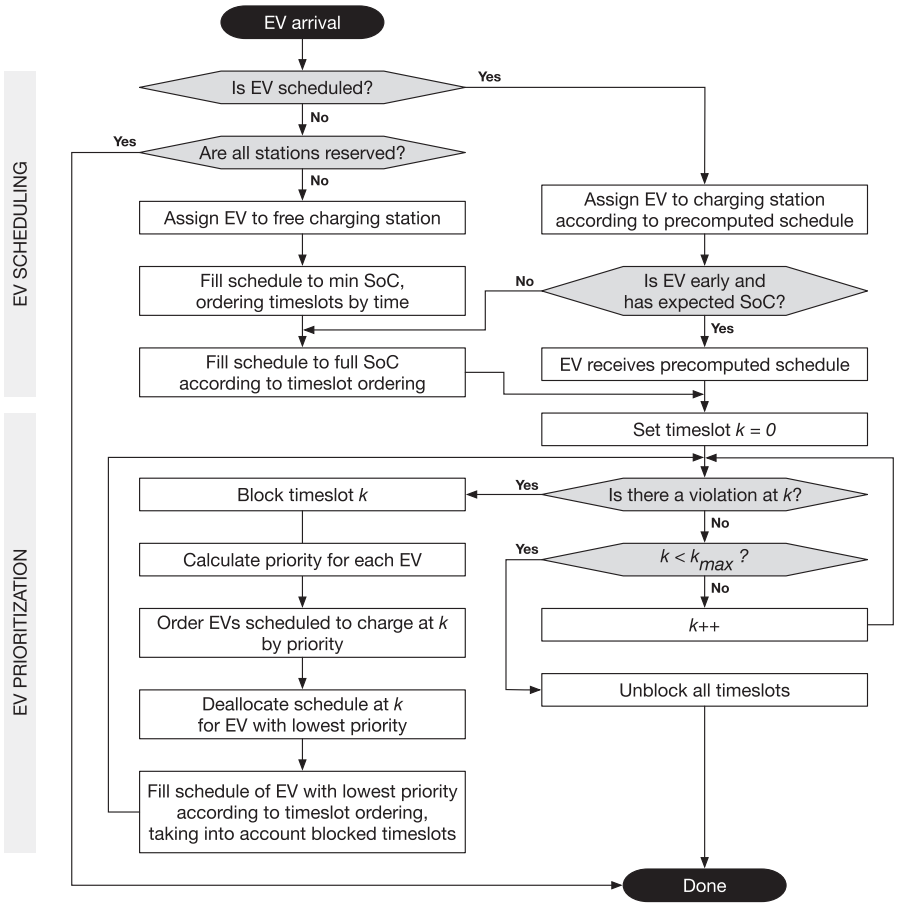

Fig. 9. Schedule-guided heuristic by Frendo et al. [20]. © IEEE.

same development effort has been applied once to our conversion routines and crypto library running in the trusted module. However, when adding a new functionality our approach only requires compiling the source program and not applying the same effort again on the new implementation.

7.2 Secure EV Charging Scheduling

In this experiment, we use DFAuth to protect sensitive data processed by a smart charging scheduler for EVs. Smart charging is a technique to schedule EV fleets making the most of existing infrastructure—for example, undersized connection lines and a limited number of charging stations.

Frendo et al. [20] present a novel approach combining day-ahead and real-time planning. Their schedule-guided heuristic takes as input a precomputed day-ahead schedule and adjusts it in real time as new information arrives. Events providing new information, for example, include EV arrival, EV departure, and price changes at energy markets. Event processing must be fast, for example, such that drivers can be assigned a charging station as they enter a parking garage. The event handling process is divided into two parts—EV scheduling and EV prioritization—as depicted in Figure 9.

To provide event responses in real time, it is convenient to utilize the cloud's powerful computing resources, high availability, and central data storage. However, sensitive data such as planned EV arrival and departure times as well as technical car properties, which could be used to identify the driver, are at risk of being revealed to the cloud service provider. We apply DFAuth such that all operations involving sensitive data are performed on encrypted data.

Experimental setup. Overall, we apply DFAuth to the schedule-guided heuristic by Frendo et al. [20] and reproduce their real-time charging simulation using our *encrypted schedule-guided heuristic*. Provided a precomputed *encrypted day-ahead schedule* as input, we simulate 1 day of events during which the current schedule has to be updated.

In particular, we apply our transformation to all classes handling sensitive data. For each car, the information protected in this way includes its minimum state of charge (min SoC), its current state of charge, and its current charging schedule. We do not protect prices at energy markets, the structure of the charging schedule (e.g., timeslot size), and the car type (i.e., whether it is a battery-electric or a plug-in hybrid electric vehicle). Results of the transformation are new classes operating on ciphertexts rather than floating-point double values. For HASE, floating-point numbers are converted to fixed-point numbers by scaling by a factor of 10^8.

The charging simulation is parameterized by the number of EVs. We consider the same parameters $\{10, 20, 30, \ldots, 400\}$ as in the original experiment [20]. The number of charging stations is constant at 25. Possible events are EV arrival, EV departure, update of expected EV arrival, update of expected EV departure, and price changes at energy markets. Each EV event occurs approximately once per EV, and new energy prices are updated every 15 minutes (i.e., 96 times per simulation).

We execute the simulation for each parameter for HASE and TACO using the same underlying data set. In each simulation, we measure the accumulated time taken to adjust the schedule as a result of an event. We distinguish the running time of code executing at the untrusted server, the time spent invoking and inside the trusted module, and the number of operations performed on encrypted data inside and outside of the trusted module. To evaluate the real-time property of our solution, we also determine the maximum event response time over all events.

Evaluation results. Figure 10 presents the total running time of each simulation for HASE and TACO. It can be seen that TACO significantly outperforms HASE in this experiment. The smallest simulation using 10 EVs takes 802 seconds for HASE but only 4 seconds for TACO. Even for the largest simulation involving 400 EVs, the running time of 139 seconds for TACO is significantly less than for 10 EVs using HASE. The graph does not include any HASE results for parameters larger than 20, because we aborted the HASE experiment due to its long running time and high response time. In Figure 10, we can also see that sometimes the running time of a simulation decreases although the number of EVs is increased (e.g., for parameters 220 and 230). This is caused by randomized sampling of simulation data, which may result in larger instances being easier to solve by the heuristic than smaller instances.

Figure 11 shows the number of trusted module calls for each simulation. The trusted module calls linearly increase in the number of EVs, as is expected based on the algorithm. For parameters 220 and 230, the number of trusted module calls are in line with the respective running times and again suggest that the simulation involving 230 EVs was easier to solve. As in the neural network experiment, HASE requires less trusted module calls than TACO. For example, for 10 EVs, HASE requires 284,290 trusted module calls, whereas TACO performs 309,656. This may result in better overall HASE performance in case an alternative trusted module with more expensive calls is used.

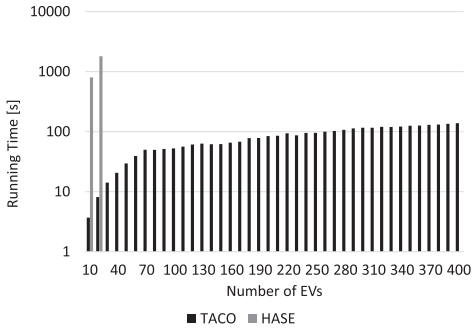

Fig. 10. Running time (in seconds) of simulations for HASE and TACO.

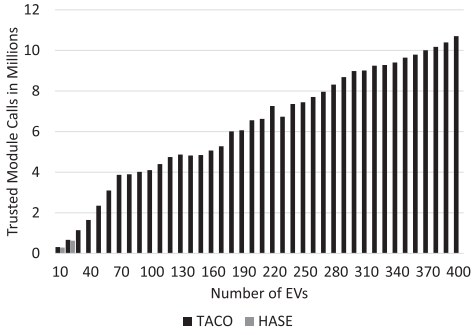

Fig. 11. Number of trusted module calls for HASE and TACO in simulations.

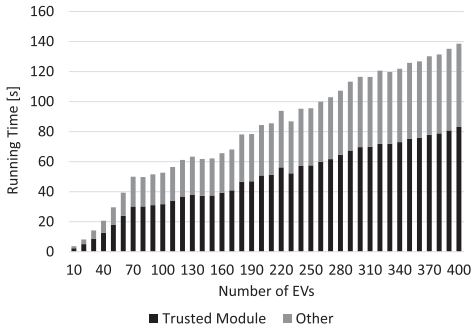

Fig. 12. Running time (in seconds) spent inside and outside the trusted module for TACO simulations.

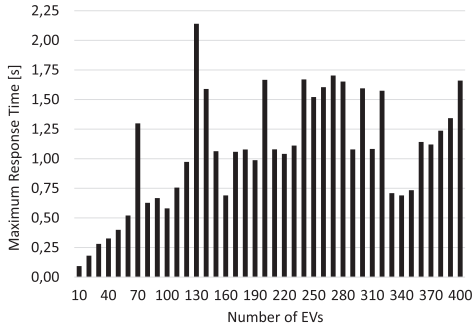

Fig. 13. Maximum event response time (in seconds) for TACO simulations.

Figure 12 illustrates the time TACO spent inside and outside the trusted module. The portion of time inside the trusted module varies between 59.5% for 320 EVs and 62.4% for 10 EVs, and as such can be considered independent of the number of EVs. In comparison, for HASE the average portion of time inside the trusted module is 99.5%, despite the lower number of trusted module calls and higher total running time.

Figure 13 presents the maximum event response time, an important property of real-time applications. By definition, the maximum response time is dominated by single events that are complex to handle for the heuristic. Such events cause spikes in the chart, as can be observed for 70 and 130 EVs. If we ignore those outliers, we can split the graph into two parts. From 10 to 90 EVs, the maximum response time continuously increases with the number of EVs. This is because time is mostly spent in the scheduling step and only a small number of violations have to be resolved in the prioritization step (cf. Figure 9). For larger numbers of EVs, the maximum response time varies between 0.70 and 2.14 seconds, an average of 1.06 seconds. Considering the response time of different simulations, we can see that it is practially independent of the number of EVs.

In summary, DFAuth using TACO enables the protection of sensitive data while simultaneously providing sufficiently fast response times required by the smart charging use case.

8 RELATED WORK

Our work is related to obfuscation techniques and trusted hardware, (homomorphic) authenticated encryption, and computation on encrypted data—including but not limited to homomorphic encryption.

Obfuscation techniques and trusted hardware. Approaches straightening or obfuscating the control flow can be combined with DFAuth on the unprotected program part and are hence mostly orthogonal to our work. We provide a detailed analysis of the security and performance implications of executing control-flow driven programs in related work [19].

Molnar et al. [39] eliminate control-flow side channels by transforming code containing conditional instructions into straight-line code employing masking.

GhostRider [37] enables privacy-preserving computation in the cloud assuming a remote trusted processor. It defends against memory side channels by obfuscating programs such that their memory access pattern is independent of control-flow instructions. However, as a hardware-software co-design, GhostRider requires a special co-processor. In contrast, our approach works on commodity SGX-enabled CPUs and provides a program-independent TCB inside the secure hardware. Raccoon [44] extends these protections to the broader class of side channels carrying information over discrete bits. Essentially, Raccoon executes both paths of a conditional branch and later combines the real and the decoy path using an oblivious store operation.

HOP [40] obfuscates programs by encrypting them such that only a trusted processor can decrypt and run them. By incorporating encryption routines into the program, HOP can be extended to also protect program input and output. However, HOP assumes the program is free of software vulnerabilities and runs the entire program inside the trusted hardware. In contrast, in DFAuth, vulnerabilities are confined to the untrusted program and the code inside the trusted module is program independent.

(Homomorphic) authenticated encryption. Authenticated encryption is an encryption mode that provides confidentiality as well as authenticity (unforgeability) and is the recommended security notion for symmetric encryption schemes. An authenticated encryption can be obtained by composing an IND-CPA secure encryption scheme with a signature or MAC [8]. Hence, one can obtain a *homomorphic* authenticated encryption by combining a homomorphic encryption scheme with a homomorphic MAC. However, since the best known homomorphic MACs [22] are not yet fully homomorphic, a different construction is required. Joo and Yun [30] provide the first fully homomorphic AE. However, their decryption algorithm is as complex as the evaluation on ciphertexts undermining the advantages of an encrypted program—that is, one could do the entire computation in the trusted module. In parallel work, Barbosa et al. [6] develop labeled homomorphic encryption that, however, has not been applied to trusted modules.

Boneh et al. [11] introduced linearly homomorphic signatures and MACs to support the efficiency gain by network coding. However, their signatures were still deterministic, hence not achieving IND-CPA security. Catalano et al. [14] integrated MACs into efficient, linearly homomorphic Paillier encryption [42] and used identifiers to support public verifiability (i.e., verification without knowledge of the plaintext). However, their scheme also has linear verification time undermining the advantages of a small trusted module.

In our HASE construction, we aimed for using identifiers and not plaintext values to enable dataflow authentication. Furthermore, we split verification into a precomputed derivation phase and a verification phase. Hence, we can achieve constant time verification.

Aggregate MACs [31] provide support for aggregation of MACs from distinct keys. However, our current dataflow authentication considers one client and secret key.

Computation on encrypted data. Since FHE [23] entails rather high computational overhead [24], researchers have resorted to partially encrypting computations. MrCrypt [49] infers feasible encryption schemes using type inference. In addition to homomorphic encryption, MrCrypt makes use of randomized and deterministic order-preserving encryption. However, the set of feasible

programs is limited and the authors only evaluate it on shallow MapReduce program snippets. Even in this case, several test cases cannot be executed. JCrypt [15] improved the type inference algorithm to a larger set of programs. Even then, no conversions between encryption schemes were performed. AutoCrypt [50] used these conversions, however, and realized their security implications. The authors hence disallowed any conversion from homomorphic encryption to searchable encryption. This restriction prevents any program from running that modifies its input and then performs a control-flow decision. Such programs include the arithmetic computations we performed in our evaluation.

Next to programs written in imperative languages (e.g., Java), programs in declarative languages (e.g., SQL) are amenable to encrypted computation. In these languages, the programmer does not specify the control-flow decisions, but they may be optimized by the interpreter or compiler. Hence, any resulting data is admissible and weaker encryption schemes must be used. Hacigümüs et al. [27] used deterministic encryption to implement a large subset of SQL. Popa et al. [43] used also randomized and order-preserving encryption in an adjustable manner [43].

Verifiable computation [21] can be used by a client to check whether a server performed a computation as intended—even on encrypted data. However, this does not prevent the attacks by malicious adversaries considered in this article. It only proves that the server performed one correct computation, but not that it did not perform any others.

Functional encryption [12] is a more powerful computation on encrypted data than homomorphic encryption. It not only can compute any function but also can reveal the result of the computation and not only its ciphertext. However, generic constructions [26] are even slower than homomorphic encryption. Searchable encryption [48] is a special case of functional encryption for comparisons. It could be used to implement comparisons in dataflow authentication. However, since the actual comparison time is so insignificant compared to the cryptographic operations, it is more efficient to implement comparison in the trusted module as well.

9 CONCLUSION

Summary. We introduce the concept of dataflow authentication (DFAuth), which prevents an active adversary from deviating from the dataflow in an outsourced program. This in turn allows safe use of re-encryptions between homomorphic and leaking encryption schemes to allow a larger class of programs to run on encrypted data where only the executed control flow is leaked to the adversary. Our implementation of DFAuth uses two novel schemes—HASE and TACO—and trusted modules in an SGX enclave. Compared to an implementation solely on FHE, our approach provides high efficiency and actually practical performance due to fast ciphertext operations and support for control flow. Compared to an implementation solely on Intel's SGX, we offer a much smaller TCB independent of the protected application. We underpin these results by an implementation of a bytecode-to-bytecode compiler that translates Java programs into Java programs operating on encrypted data using DFAuth.

Future work. First, our security model does not assume the existence of verification oracles for MACs. There exist schemes that are secure in our model, but not with verification oracles [7]. We leave the construction of a proof in a security model with verification models as future work. Second, DFAuth considers a simple scenario between two parties: a client and a server (which has a trusted module). In the context of extending DFAuth to multiple clients, an open question is whether HASE can be adjusted to a public-key setting without weakening its security properties. Third, the trusted module in this work was implemented using an SGX enclave. An open question is whether DFAuth can gain any security or performance improvements from using alternative trusted execution environments.

APPENDICES

A POSTPONED SECURITY REDUCTIONS

We perform security reductions using a *sequence of games*. The first game is the original security experiment provided by the security definition. Each subsequent game is equal to the previous game except for some small well-defined change for which we argue that it does only negligibly influence adversarial advantage. The last game then has a special and easy to verify property (e.g., the adversary has no advantage over a blind guess). Only negligible change in advantage between subsequent games implies only negligible change in advantage between the first and the last game, which concludes the reduction.

A.1 Proof of Theorem 1 (HASE-IND-CPA)

Proof. Let $\Pi, \mathcal{G}, H$ be as described, and let $\mathcal{A}$ be a PPT adversary. We use a sequence of games to show that $\mathcal{A}$'s advantage $\mathrm{Adv}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(\lambda)$ is negligible in λ. For Game n, we use S_n to denote the event that $b = b'$. The final game and the encryption oracle used in all games are given in Figure 14.

Game 0. This is the original experiment from Definition 3, but instead of relying on Π, the challenger performs the exact same computations on its own. Clearly, $\mathrm{Adv}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(\lambda) = |\Pr[S_0] - \frac{1}{2}|$.

Game 1 (Indistinguishability-Based Transition). Instead of deriving the label used in the third component of the challenge ciphertext using the PRF $H : \mathcal{K} \times \mathcal{I} \to \mathbb{G}$ for some random $k \leftarrow\!\!\$\, \mathcal{K}$, we make use of a random function $f \leftarrow\!\!\$\, \mathcal{F}$ from the set of functions $\mathcal{F} = \{F : \mathcal{I} \to \mathbb{G}\}$.

We construct a polynomial time algorithm $\mathcal{B}$ distinguishing between a PRF (for a random key) and a random function using $\mathcal{A}$ as a black box. If $\mathcal{B}$'s oracle is a PRF, then the view of $\mathcal{A}$ is distributed as in Game 0 and we have $\Pr[S_0] = \Pr\left[\mathcal{B}^{\mathcal{A},H(k,\cdot)}(1^\lambda) = 1\right]$ for some $k \leftarrow\!\!\$\, \mathcal{K}$. If $\mathcal{B}$'s oracle is a random function, then the view of $\mathcal{A}$ is distributed as in Game 1 and thus we have $\Pr[S_1] = \Pr\left[\mathcal{B}^{\mathcal{A},f(\cdot)}(1^\lambda) = 1\right]$ for some $f \leftarrow\!\!\$\, \mathcal{F}$. Under the assumption that H is a PRF, $|\Pr[S_0] - \Pr[S_1]|$ is negligible.

Game 2 (Conceptual Transition). Because f is only evaluated on a single input i_b and f is a random function, the result is a random element of $\mathcal{G}$. Thus, instead of computing $l := f(i_b)$, we can compute $l := g^s$ for a random exponent $s \leftarrow\!\!\$\, \mathbb{Z}_q$. Since this is only a conceptual change, we have $\Pr[S_1] = \Pr[S_2]$.

Game 3 (Indistinguishability-Based Transition). In the challenge ciphertext, we replace $h^r = g^{xr}$ with a random group element g^z generated by raising g to the power of a random $z \leftarrow\!\!\$\, \mathbb{Z}_q$.

We construct a polynomial time distinguishing algorithm $\mathcal{D}$ solving the DDH problem that interpolates between Game 2 and Game 3. If $\mathcal{D}$ receives a real triple $(g^\alpha, g^\beta, g^{\alpha\beta})$ for $\alpha, \beta \leftarrow\!\!\$\, \mathbb{Z}_q$, then $\mathcal{A}$ operates on a challenge ciphertext constructed as in Game 2 and thus we have

$$\Pr[S_2] = \Pr\left[\mathcal{D}^{\mathcal{A}}(\mathbb{G}, q, g, g^\alpha, g^\beta, g^{\alpha\beta}) = 1\right].$$

If $\mathcal{D}$ receives a random triple $(g^\alpha, g^\beta, g^\gamma)$ for $\alpha, \beta, \gamma \leftarrow\!\!\$\, \mathbb{Z}_q$, then $\mathcal{A}$ operates on a challenge ciphertext constructed as in Game 3 and thus we have

$$\Pr[S_3] = \Pr\left[\mathcal{D}^{\mathcal{A}}(\mathbb{G}, q, g, g^\alpha, g^\beta, g^\gamma) = 1\right].$$

In both cases, $\mathcal{D}$ receives $(\mathbb{G}, q, g)$ output by $\mathcal{G}(1^\lambda)$. Under the assumption that the DDH problem is hard relative to $\mathcal{G}$, $|\Pr[S_2] - \Pr[S_3]|$ is negligible.

$\underline{\text{Game}_3^{\text{IND-CPA}}_{\mathcal{A}}(\lambda)}$

$(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^\lambda)$

$a, x, y \leftarrow\$ \mathbb{Z}_q, k \leftarrow\$ \mathcal{K}$

$h := g^x, j := g^y$

$ek := \mathbb{G}$

$sk := (\mathbb{G}, q, g, a, x, y, h, j, k)$

$(m_0, m_1, i_0, i_1, \text{st}) \leftarrow \mathcal{A}^{E(sk,\cdot,\cdot)}(1^\lambda, ek)$

$b \leftarrow\$ \{0, 1\}$

$r, \boxed{s}, \boxed{z} \leftarrow\$ \mathbb{Z}_q$

$l := \boxed{g^s}$

$c := (g^r, \boxed{g^z} \cdot m_b, j^r \cdot {m_b}^a \cdot l)$

$b' \leftarrow \mathcal{A}^{E(sk,\cdot,\cdot)}(1^\lambda, c, \text{st})$

return $b = b'$

$\underline{E(sk, m, i)}$

parse $sk = (\mathbb{G}, q, g, a, x, y, h, j, k)$

$r \leftarrow\$ \mathbb{Z}_q$

$l := H(k, i)$

$c := (g^r, h^r \cdot m, j^r \cdot m^a \cdot l)$

return c

Fig. 14. Final security experiment used in the HASE-IND-CPA proof. Changes compared to the first experiment are highlighted.

Conclusion. In the last game, the first component of the challenge ciphertext is trivially independent of the challenge plaintext as well as the challenge identifier. In the second component, g^z acts like a one-time pad and completely hides m_b. Similarly, $l = g^s$ acts like a one-time pad in the third component. Because the challenge ciphertext does not contain any information about m_b or i, we conclude that $\Pr[S_3] = \frac{1}{2}$. Overall, we have that $\text{Adv}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(\lambda) = \text{negl}(\lambda)$. □

A.2 Proof of Theorem 2 (HASE-UF-CPA)

Proof. Let $\Pi, \mathcal{G}, H$ be as described, and let $\mathcal{A}$ be a PPT adversary. We use a sequence of games to show that $\mathcal{A}$'s advantage $\text{Adv}^{\text{UF-CPA}}_{\mathcal{A},\Pi}(\lambda)$ is negligible in λ. For Game n, we use S_n to denote the event that the adversary wins the game. The final game is illustrated in Figure 15.

Game 0. This is the original experiment from Definition 4, but instead of relying on Π, the challenger performs the exact same computations on its own. Clearly, $\text{Adv}^{\text{UF-CPA}}_{\mathcal{A},\Pi}(\lambda) = |\Pr[S_0]|$.

Game 1 (Conceptual Transition). We eliminate the conditional statement by comparing t and w in the return statement.

Game 2 (Indistinguishability-Based Transition). We replace the PRF $H(k, \cdot)$ with a function $f(\cdot)$ chosen at random. Under the assumption that H is a PRF, we have that $|\Pr[S_1] - \Pr[S_2]|$ is negligible as in the previous security reduction in Theorem 1.

Conclusion. We show that $\Pr[S_2] = \text{negl}(\lambda)$. Let X be the event that $\forall i \in I : \exists (m, i) \in S$ (i.e., all identifiers have been used in encryption oracle queries).

In case event X does not happen, the challenger evaluates function f on at least one new argument. By the definition of f, the result is a random value in the image of f. This random group element acts as a one-time pad and makes l look random. Subsequently, t is also random from the point of view of the adversary. To win the experiment, $\mathcal{A}$ has to fulfill $t = w$. Because t is random, $\mathcal{A}$ cannot guess the correct w with probability better than $\frac{1}{q}$. Thus, we have

$$\Pr[S_2 \wedge \neg X] = \frac{1}{q} \cdot \Pr[\neg X]. \tag{1}$$

$\underline{\text{Game}_2^{\text{UF-CPA}}_{\mathcal{A},\mathcal{G},H}(\lambda)}$

$S := \{\}$

$(\mathbb{G}, q, g) \leftarrow \mathcal{G}(1^\lambda)$

$a, x, y \leftarrow\$ \mathbb{Z}_q, k \leftarrow\$ \mathcal{K}, \boxed{f \leftarrow\$ \mathcal{F}}$

$h := g^x, j := g^y$

$ek := \mathbb{G}$

$sk := (\mathbb{G}, q, g, a, x, y, h, j, \boxed{f})$

$(c, I) \leftarrow \mathcal{A}^{E(sk,\cdot,\cdot)}(1^\lambda, ek)$

$l := \prod_{i \in I} \boxed{f(i)}$

parse $c = (u, v, w)$

$m := u^{-x} \cdot v$

$t := u^y \cdot m^a \cdot l$

if $I \notin \pi_2(S)$ **then**

  return $\boxed{t = w}$

else

  $\tilde{m} := \bigoplus_{(m',i) \in S, i \in I} m'$

  return $\boxed{t = w} \wedge m \neq \tilde{m}$

$\underline{E(sk, m, i)}$

parse $sk = (\mathbb{G}, q, g, a, x, y, h, j, \boxed{f})$

if $i \in \pi_2(S)$ **then**

  return $\perp$

else

  $S := S \cup \{(m, i)\}$

  $r \leftarrow\$ \mathbb{Z}_q, l := \boxed{f(i)}$

  $c := (g^r, h^r \cdot m, j^r \cdot m^a \cdot l)$

  return c

Fig. 15. Final security experiment used in the HASE-UF-CPA proof. Changes compared to the first experiment are highlighted.

Recall that q is the order of $\mathbb{G}$ (of which w is an element), and both are output by the group generation algorithm $\mathcal{G}(1^\lambda)$. Also note that $\neg X$ holds when $\mathcal{A}$ performs no encryption queries at all.

Now consider the case when event X happens, and let (c, I) be the output of the adversary. The set of identifiers I determines a label l and an expected message $\tilde{m}$. Furthermore, let $\tilde{c} = (\tilde{u}, \tilde{v}, \tilde{w})$ be the ciphertext resulting from the application of Π.Eval to ciphertexts identified by I. As $\tilde{c}$ is an honestly derived encryption of $\tilde{m}$, the following must hold:

$$\begin{aligned} \tilde{m} &= \tilde{u}^{-x} \cdot \tilde{v} \\ \tilde{w} &= \tilde{u}^y \cdot \tilde{m}^a \cdot l \\ &= (\tilde{u}^{y-x} \cdot \tilde{v})^a \cdot l \end{aligned} \tag{2}$$

Similarly, for $c = (u, v, w)$ to be accepted as a forgery regarding I, it must hold that

$$w = (u^{y-x} \cdot v)^a \cdot l \tag{3}$$

for some $m := u^{-x} \cdot v \neq \tilde{m}$. Because $m \neq \tilde{m}$, we know that $\tilde{u}^{y-x} \cdot \tilde{v} \neq u^{y-x} \cdot v$ and $\tilde{w} \neq w$.

Combining Equations (2) and (3) yields

$$\begin{aligned} \frac{\tilde{w}}{w} &= \frac{(\tilde{u}^{y-x} \cdot \tilde{v})^a \cdot l}{(u^{y-x} \cdot v)^a \cdot l} \\ &= \left(\frac{\tilde{u}^{y-x} \cdot \tilde{v}}{u^{y-x} \cdot v}\right)^a . \end{aligned} \tag{4}$$

For c to be a forgery with regard to I, Equation (4) needs to be satisfied. But since a is a random element of $\mathbb{Z}_q$, the probability that $\mathcal{A}$ can satisfy (4) is only $\frac{1}{q}$. Hence,

$$\Pr[S_2 \wedge X] = \frac{1}{q} \cdot \Pr[X]. \tag{5}$$

Summing up (1) and (5), we have

$$\Pr[S_2] = \Pr[S_2 \wedge \neg X] + \Pr[S_2 \wedge X] = \frac{1}{q},$$

and, overall we have that $\mathsf{Adv}^{\text{UF-CPA}}_{\mathcal{A},\Pi}(\lambda) = \mathsf{negl}(\lambda)$. □

A.3 Theorems 3 and 4

The security of Construction 2 (Theorem 3 and Theorem 4) follows directly from the security of Construction 1 (Theorem 1 and Theorem 2).

A.4 Proof of Theorem 5 (TACO-IND-CPA)

Proof. Let Π be Construction 3 and SE its symmetric encryption scheme. Note that if SE has IND-CCA security, it also has IND-CPA security.

Assume there exists an adversary winning the TACO-IND-CPA game with non-negligible probability. By definition, the adversary can then distinguish for chosen messages $m_1, m_2 \in \mathcal{M}$ and identifiers $i_1, i_2 \in \mathcal{I}$ the ciphertext

$$c_1 \leftarrow \Pi.\mathsf{Enc}(sk, m_1, i_1) \stackrel{\text{Def}}{=} SE.\mathsf{Enc}(sk, m_1 \| \mathsf{Der}(i_1))$$

from

$$c_2 \leftarrow \Pi.\mathsf{Enc}(sk, m_2, i_2) \stackrel{\text{Def}}{=} SE.\mathsf{Enc}(sk, m_2 \| \mathsf{Der}(i_2))$$

with non-negligible probability. In case the adversary performed no evaluation oracle queries, this is a contradiction to IND-CPA security of SE.

To show that the evaluation oracle provides only negligible advantage to the adversary, we simulate it as an attacker on the IND-CCA property of the SE scheme. The IND-CCA property of the SE scheme provides indistinguishability with additional access to a decryption oracle. Assume the adversary wants to perform an evaluation φ on ciphertexts $C = (c_1, \ldots, c_{p_\varphi})$. In case C does not include the challenge ciphertext c, the evaluation can be simulated by decrypting each ciphertext using the SE.Dec oracle, calculating φ on the results, and encrypting with the encryption oracle again. In case the adversary wants to perform an evaluation φ on a tuple including the challenge ciphertext c, instead of computing $SE.\mathsf{Dec}(sk, c)$, we use m_b and i_b with $b \leftarrow\!\!\$\, \{0, 1\}$ for the calculation of φ. If the attacker can distinguish the simulation from a honest oracle with non-negligible probability, he can again distinguish $SE.\mathsf{Enc}(m'_1)$ from $SE.\mathsf{Enc}(m'_2)$ for at least two $m'_1, m'_2 \in \mathcal{M}$ with non-negligible probability. Since the number of evaluations requested by the attacker is polynomial in λ, we can show that Π is TACO-IND-CPA secure. □

A.5 Proof of Theorem 6 (TACO-UF-CPA)

Proof. Let Π be Construction 3, SE its symmetric encryption scheme, and $\Omega = (\mathsf{Gen}, H)$ its hash function. The instructions defined in the TACO-UF-CPA game lead directly to the following

equalities:

$$\begin{aligned}
&\mathrm{Adv}^{\text{UF-CPA}}_{\mathcal{A},\Pi}(\lambda)\\
&= \Pr[m_1 \neq \perp \wedge m_2 \neq \perp \wedge m_1 \neq m_2]\\
&= \Pr[\Pi.\mathrm{Dec}(sk, c_1, l) \neq \perp \wedge \Pi.\mathrm{Dec}(sk, c_2, l) \neq \perp \wedge m_1 \neq m_2]\\
&= \Pr[SE.\mathrm{Dec}(sk, c_1) \neq \perp \wedge l_1 = l \wedge SE.\mathrm{Dec}(sk, c_2) \neq \perp \wedge l_2 = l \wedge m_1 \neq m_2]\\
&\leq \Pr[l_1 = l_2 \wedge SE.\mathrm{Dec}(sk, c_1) \neq \perp \wedge SE.\mathrm{Dec}(sk, c_2) \neq \perp \wedge m_1 \neq m_2]. \qquad (6)
\end{aligned}$$

Here, l_1 and l_2 are the parsed labels of the back part of the $SE.\mathrm{Dec}$ decrypted ciphertexts. Applying the definition of conditional probability results in

$$\mathrm{Adv}^{\text{UF-CPA}}_{\mathcal{A},\Pi}(\lambda) \leq \Pr[l_1 = l_2 \mid m_1 \neq m_2 \wedge SE.\mathrm{Dec}(sk, c_1) \neq \perp \wedge SE.\mathrm{Dec}(sk, c_2) \neq \perp]. \qquad (7)$$

Since the underlying encryption scheme is assumed to be unforgeable, for $SE.\mathrm{Dec}(sk, c_1) \neq \perp \wedge SE.\mathrm{Dec}(sk, c_2) \neq \perp$ to hold with non-negligible probability, the ciphertexts must have been created using the secret key. This implies that the ciphertexts were created by either the evaluation oracle or the encryption oracle. For the evaluation not to result in $\perp$, it must be performed on valid ciphertexts. Let $\Pr[S_n]$ be the probability in Inequality (7), where n is the number of evaluations an adversary performs on valid ciphertexts to create the ciphertexts c_1 and c_2 before passing them to the challenger. We show the security by an induction proof over n:

- *Claim*: $\Pr[S_n]$ is negligible $\forall n \in \mathbb{N}$ with n polynomial in λ.
- *First base case for* $n = 0$: No evaluations have been performed such that

$$\begin{aligned}
\Pr[S_0] &= \Pr[l_1 = l_2 \mid m_1 \neq m_2 \wedge c_1 = E(sk, m_1, i_1) \wedge c_2 = E(sk, m_2, i_2)]\\
&= \Pr[H(k, i_1 \| 0) = H(k, i_2 \| 0) \mid m_1 \neq m_2 \wedge c_1 = E(sk, m_1, i_1) \wedge c_2 = E(sk, m_2, i_2)].
\end{aligned}$$

  Since the encryption oracle does not allow to encrypt different messages with the same identifier, it must hold that $i_1 \neq i_2$, which leads to

$$\Pr[S_0] = \Pr[H(k, i_1 \| 0) = H(k, i_2 \| 0) \mid i_1 \neq i_2].$$

  This probability is negligible due to the assumption that Ω is collision resistant.
- *Second base case for* $n = 1$: Without loss of generality, let c_1 be the ciphertext created by an evaluation of an operation φ and valid ciphertexts $(c_{1,1}, \ldots, c_{1,p_\varphi})$ with corresponding labels $(l_{1,1}, \ldots, l_{1,p_\varphi})$. Then

$$\begin{aligned}
\Pr[S_1] &= \Pr[l_1 = l_2 \mid m_1 \neq m_2 \wedge c_1 = \mathrm{Eval}(sk, \varphi, (c_{1,1}, \ldots, c_{1,p_\varphi})) \wedge c_2 = E(sk, m_2, i_2)]\\
&= \Pr\left[H(k, \mathtt{id}(\varphi) \| l_{1,1} \| \ldots \| l_{1,p_\varphi} \| 1) = H(k, i_2 \| 0)\right].
\end{aligned}$$

  This probability is negligible due to the assumption that Ω is collision resistant.
- *Step case for* $n \rightsquigarrow n+1$: Assume the claim is true for $0 \leq n^* \leq n$. If only one challenger ciphertext is created by evaluations, then the proof in the second base case applies. Otherwise, for $j \in \{0, 1\}$, let c_j be the ciphertext created by an evaluation of operation φ_j and valid ciphertexts $(c_{j,1}, \ldots, c_{j,p_{\varphi_j}})$ with corresponding labels $(l_{j,1}, \ldots, l_{j,p_{\varphi_j}})$ and corresponding messages

$(m_{j,1}, \ldots, m_{j,p_{\varphi_j}})$. Then

$$\begin{aligned}
\Pr[S_{n+1}] &= \Pr\,[l_1 = l_2 \mid m_1 \neq m_2 \\
&\qquad \wedge \forall j \in \{0,1\} : c_j = \mathsf{Eval}(sk, \varphi_j, (c_{j,1}, \ldots, c_{j,p_{\varphi_j}})) \\
&\qquad \wedge SE.\mathsf{Dec}(sk, c_1) \neq \perp \wedge SE.\mathsf{Dec}(sk, c_2) \neq \perp] \\
&= \Pr\,[H(k, \mathtt{id}(\varphi_1) \| l_{1,1} \| \ldots \| l_{1,p_{\varphi_1}} \| 1) \\
&= \qquad H(k, \mathtt{id}(\varphi_2) \| l_{2,1} \| \ldots \| l_{2,p_{\varphi_2}} \| 1) \mid m_1 \neq m_2 \\
&\qquad\quad \wedge SE.\mathsf{Dec}(sk, c_1) \neq \perp \wedge SE.\mathsf{Dec}(sk, c_2) \neq \perp] \\
&= \Pr\,[H(\ldots) = H(\ldots) \wedge (\varphi_1 \neq \varphi_2 \vee \exists j : l_{1,j} \neq l_{2,j}) \mid m_1 \neq m_2 \\
&\qquad \wedge SE.\mathsf{Dec}(sk, c_1) \neq \perp \wedge SE.\mathsf{Dec}(sk, c_2) \neq \perp], \qquad (8) \\
&\qquad + \Pr\,[H(\ldots) = H(\ldots) \wedge (\varphi_1 = \varphi_2 \wedge \forall j : l_{1,j} = l_{2,j}) \mid m_1 \neq m_2 \\
&\qquad \wedge SE.\mathsf{Dec}(sk, c_1) \neq \perp \wedge SE.\mathsf{Dec}(sk, c_2) \neq \perp]. \qquad (9)
\end{aligned}$$

Probability (8) is negligible since the input to H differs and Ω is collision resistant. We explore Probability (9) in more detail by considering its conditions.

$$\begin{aligned}
& m_1 \neq m_2 \\
& \Rightarrow SE.\mathsf{Dec}(sk, \mathsf{Eval}(sk, \varphi_1, (c_{1,1}, \ldots, c_{1,p_{\varphi_1}}))) \neq SE.\mathsf{Dec}(sk, \mathsf{Eval}(sk, \varphi_2, (c_{2,1}, \ldots, c_{2,p_{\varphi_2}}))) \\
& \Rightarrow \varphi_1(SE.\mathsf{Dec}(sk, c_{1,1}), \ldots, SE.\mathsf{Dec}(sk, c_{1,p_{\varphi_1}})) \neq \varphi_2(SE.\mathsf{Dec}(sk, c_{2,1}), \ldots, SE.\mathsf{Dec}(sk, c_{2,p_{\varphi_2}})) \\
& \Rightarrow \varphi_1(SE.\mathsf{Dec}(sk, c_{1,1}), \ldots, SE.\mathsf{Dec}(sk, c_{1,p_{\varphi_1}})) \neq \varphi_1(SE.\mathsf{Dec}(sk, c_{2,1}), \ldots, SE.\mathsf{Dec}(sk, c_{2,p_{\varphi_1}})) \\
& \Rightarrow \exists j^* : SE.\mathsf{Dec}(sk, c_{1,j^*}) \neq SE.\mathsf{Dec}(sk, c_{2,j^*}) \qquad (10)
\end{aligned}$$

We choose such an j^* fulfilling Inequality (10) and obtain the following.

$$\begin{aligned}
\Pr[S_{n+1}] &= \Pr\,[H(\ldots) = H(\ldots) \wedge \varphi_1 = \varphi_2 \wedge \forall j : l_{1,j} = l_{2,j} \mid m_1 \neq m_2 \\
&\qquad \wedge SE.\mathsf{Dec}(sk, c_1) \neq \perp \wedge SE.\mathsf{Dec}(sk, c_2) \neq \perp] + negl(\lambda) \\
&\leq \Pr\,[H(\ldots) = H(\ldots) \wedge \varphi_1 = \varphi_2 \wedge l_{1,j^*} = l_{2,j^*} \mid m_1 \neq m_2 \\
&\qquad \wedge SE.\mathsf{Dec}(sk, c_1) \neq \perp \wedge SE.\mathsf{Dec}(sk, c_2) \neq \perp] + negl(\lambda) \\
&\leq \Pr\,[l_{1,j^*} = l_{2,j^*} \mid m_1 \neq m_2 \\
&\qquad \wedge SE.\mathsf{Dec}(sk, c_1) \neq \perp \wedge SE.\mathsf{Dec}(sk, c_2) \neq \perp] + negl(\lambda) \\
&= \Pr\,[l_{1,j^*} = l_{2,j^*} \mid m_1 \neq m_2 \\
&\qquad \wedge SE.\mathsf{Dec}(sk, c_1) \neq \perp \wedge SE.\mathsf{Dec}(sk, c_2) \neq \perp \\
&\qquad \wedge SE.\mathsf{Dec}(sk, c_{1,j^*}) \neq SE.\mathsf{Dec}(sk, c_{2,j^*})] + negl(\lambda) \\
&\leq \Pr\,[l_{1,j^*} = l_{2,j^*} \mid SE.\mathsf{Dec}(sk, c_1) \neq \perp \\
&\qquad \wedge SE.\mathsf{Dec}(sk, c_2) \neq \perp \wedge m_{1,j^*} \neq m_{2,j^*}] + negl(\lambda)
\end{aligned}$$

Moreover, it holds the following.

$$\begin{aligned}
& SE.\mathsf{Dec}(sk, c_1) \neq \perp \wedge SE.\mathsf{Dec}(sk, c_2) \neq \perp \\
& \Rightarrow SE.\mathsf{Dec}(\mathsf{Eval}(sk, \varphi_1, (c_{1,1}, \ldots, c_{1,p_{\varphi_1}}))) \neq \perp \wedge SE.\mathsf{Dec}(\mathsf{Eval}(sk, \varphi_2, (c_{2,1}, \ldots, c_{2,p_{\varphi_2}}))) \neq \perp \\
& \Rightarrow SE.\mathsf{Dec}(c_{1,j^*}) \neq \perp \wedge SE.\mathsf{Dec}(c_{2,j^*}) \neq \perp
\end{aligned}$$

$$\begin{aligned}
\Pr[S_{n+1}] &\leq \Pr\,[l_{1,j^*} = l_{2,j^*} \mid SE.\mathsf{Dec}(c_{1,j^*}) \neq \perp \wedge SE.\mathsf{Dec}(c_{2,j^*}) \neq \perp \wedge m_{1,j^*} \neq m_{2,j^*}] + negl(\lambda) \\
&= \Pr[S_{n^*}] + negl(\lambda), \qquad n^* \leq n
\end{aligned}$$

This probability is negligible according to the induction hypothesis proving that that $\forall n \in \mathbb{N}\ \Pr[S_n]$ is negligible. Because the number of evaluations an adversary can perform n is polynomial in λ, $\mathrm{Adv}^{\text{UF-CPA}}_{\mathcal{A},\Pi}(\lambda)$ is negligible. □

B ADDITIVE HASE BENCHMARK

The decryption algorithm of the Additive HASE construction (Construction 2) involves computing a discrete logarithm for each ciphertext component (after Elgamal decryption). Since each homomorphic addition can increase the bit length of exponents by 1, a large amount of homomorphic additions can make decryption exponentially costlier (or impossible, assuming only a lookup table with precomputed logarithms), despite the use of the CRT approach provided by Hu et al. [29]. In the following experiment, we demonstrate that re-encryptions inserted by the DFAuth compiler are an effective measure for preventing excessive exponent growth and ensuring an efficient decryption.

Experimental setup. Thoughout this experiment, we use a CRT decomposition involving 13 different 5-bit primes. These parameters were chosen such that we can represent 64-bit integer values. In the trusted module, a lookup table containing 2^{18} precomputed discrete logarithms was generated. We measure the runtime of the decryption algorithm when applied to a ciphertext resulting from the homomorphic evaluation of n ciphertexts. For each n, we consider two variants of the experiment: one without re-encryptions, and the other with re-encryptions performed after every 4,000 homomorphic evaluations. We use $n \in \{10, 100, 1{,}000, 10{,}000, 100{,}000\}$ and perform each measurement 100 times.

Evaluation results. Figure 16 presents the mean runtime of the decryption algorithm for each n and each variant (without re-encryption and with re-encryptions). We can see that the decryption time without re-encryptions are mostly constant up to 1,000 homomorphic evaluations but increases drastically for larger numbers of evaluations. The reason for this sharp increase in decryption time is likely the fact that the discrete logarithms can no longer be computed via table lookup but the decryption has to fall back to exhaustive search (cf. Section 6.1). In comparison, when re-encryptions are performed, the decryption time only increases minimally, even for $n = 100{,}000$.

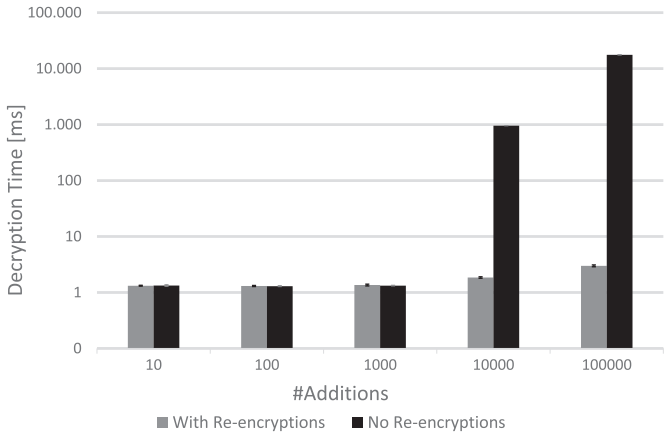

Fig. 16. Mean runtime (in milliseconds) to decrypt a ciphertext produced by summing up a varying number of ciphertexts using the Additive HASE scheme.

REFERENCES

[1] MPIR. n.d. MPIR: Multiple Precision Integers and Rationals. Retrieved February 23, 2022 from http://mpir.org.

[2] Neuroph. n.d. Neuroph—Java Neural Network Framework. Retrieved February 23, 2022 from http://neuroph.sourceforge.net.

[3] Libsodium. n.d. The Sodium Crypto Library (libsodium). Retrieved February 23, 2022 from https://download.libsodium.org/doc/.

[4] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Detecting equality of variables in programs. In *Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL'88)*.

[5] Ittai Anati, Shay Gueron, Simon P. Johnson, and Vincent R. Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*.

[6] Manuel Barbosa, Dario Catalano, and Dario Fiore. 2017. Labeled homomorphic encryption—Scalable and privacy-preserving processing of outsourced data. In *Proceedings of the 22nd European Symposium on Research in Computer Security (ESORICS'17)*.

[7] Mihir Bellare, Oded Goldreich, and Anton Mityagin. 2004. The power of verification queries in message authentication and authenticated encryption. *IACR Cryptolology ePrint Archive*. Retrieved February 23, 2022 from http://eprint.iacr.org/2004/309.

[8] Mihir Bellare and Chanathip Namprempre. 2008. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. *Journal of Cryptology* 21, 4 (2008), 461–491.

[9] Mihir Bellare and Phillip Rogaway. 2006. Code-based game-playing proofs and the security of triple encryption. In *Proceedings of the 25th International Conference on Advances in Cryptology (EUROCRYPT'06)*.

[10] Daniel J. Bernstein. 2006. Curve25519: New Diffie-Hellman speed records. In *Proceedings of the 9th International Conference on Theory and Practice of Public-Key Cryptography, (PKC'06)*.

[11] Dan Boneh, David Freeman, Jonathan Katz, and Brent Waters. 2009. Signing a linear subspace: Signature schemes for network coding. In *Proceedings of the 12th International Workshop on Public Key Cryptography (PKC'09)*.

[12] Dan Boneh, Amit Sahai, and Brent Waters. 2011. Functional encryption: Definitions and challenges. In *Proceedings of the 8th Theory of Cryptography Conference (TCC'11)*.

[13] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure: SGX cache attacks are practical. In *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT'17)*.

[14] Dario Catalano, Antonio Marcedone, and Orazio Puglisi. 2014. Authenticating computation on groups: New homomorphic primitives and applications. In *Proceedings of the 20th International Conference on the Advances in Cryptology (ASIACRYPT'14)*.

[15] Yao Dong, Ana Milanova, and Julian Dolby. 2016. JCrypt: Towards computation over encrypted data. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform (PPPJ'16)*.

[16] D. Eastlake and T. Hansen. 2006. *US Secure Hash Algorithms (SHA and HMAC-SHA)*. RFC 4634 (Informational). IETF.

[17] Taher Elgamal. 1985. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory* 31, 4 (1985), 10–18.

[18] Andreas Fischer, Benny Fuhry, Florian Kerschbaum, and Eric Bodden. 2020. Computation on encrypted data using dataflow authentication. In *Proceedings of the 20th Privacy Enhancing Technologies Symposium (PETS'20)*. https://petsymposium.org/2020/files/papers/issue1/popets-2020-0002.pdf.

[19] Andreas Fischer, Jonas Janneck, Jörn Kußmaul, Nikolas Krätzschmar, Florian Kerschbaum, and Eric Bodden. 2020. PASAPTO: Policy-aware security and performance trade-off analysis —Computation on encrypted data with restricted leakage. In *Proceedings of the 33rd IEEE Computer Security Foundations Symposium*. IEEE, Los Alamitos, CA, 230–245.

[20] Oliver Frendo, Nadine Gaertner, and Heiner Stuckenschmidt. 2019. Real-time smart charging based on precomputed schedules. *IEEE Transactions on Smart Grid* 10, 6 (2019), 6921–6932.

[21] Rosario Gennaro, Craig Gentry, and Bryan Parno. 2011. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Proceedings of the 30th International Conference on Advances in Cryptology (CRYPTO'11)*.

[22] Rosario Gennaro and Daniel Wichs. 2013. Fully homomorphic message authenticators. In *Proceedings of the 19th International Conference on the Advances in Cryptology (ASIACRYPT'13)*.

[23] Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Symposium on Theory of Computing (STOC'09)*.

[24] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2012. Homomorphic evaluation of the AES circuit. In *Proceedings of the 32nd International Conference on Advances in Cryptology (CRYPTO'12)*.

[25] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. 2016. CryptoNets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of the 33rd International Conference on Machine Learning (ICML'16)*.

[26] Shafi Goldwasser, Yael Tauman Kalai, Raluca A. Popa, Vinod Vaikuntanathan, and Nickolai Zeldovich. 2013. Reusable garbled circuits and succinct functional encryption. In *Proceedings of the Symposium on Theory of Computing (STOC'13)*.

[27] Hakan Hacigümüş, Bala Iyer, Chen Li, and Sharad Mehrotra. 2002. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'02)*.

[28] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. 2013. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*.

[29] Yin Hu, William Martin, and Berk Sunar. 2012. Enhanced flexibility for homomorphic encryption schemes via CRT. In *Proceedings (Industrial Track) of the 10th International Conference on Applied Cryptography and Network Security (ACNS'12)*.

[30] Chihong Joo and Aaram Yun. 2014. Homomorphic authenticated encryption secure against chosen-ciphertext attack. In *Proceedings of the 20th International Conference on the Advances in Cryptology (ASIACRYPT'14)*.

[31] Jonathan Katz and Yehuda Lindell. 2008. Aggregate message authentication codes. In *Proceedings of the Cryptographers' Track of the RSA Conference (CT-RSA'08)*.

[32] Jonathan Katz and Yehuda Lindell. 2014. *Introduction to Modern Cryptography* (2nd ed.). Chapman & Hall/CRC.

[33] T. Kivinen and M. Kojo. 2003. *More Modular Exponential (MODP) Diffie-Hellman groups for Internet Key Exchange (IKE)*. RFC 3526 (Proposed Standard). IETF.

[34] Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. 2011. The Soot framework for Java program analysis: A retrospective. In *Proceedings of the Cetus Users and Compiler Infastructure Workshop (CETUS'11)*.

[35] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Chongho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. 2017. Hacking in darkness: Return-oriented programming against secure enclaves. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*.

[36] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium (USENIX Security'17)*.

[37] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. 2015. GhostRider: A hardware-software system for memory trace oblivious computation. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*.

[38] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. 2013. Innovative instructions and software model for isolated execution. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP'13)*.

[39] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. 2005. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of the 8th International Conferenceon Information Security and Cryptology (ICISC'05)*.

[40] Kartik Nayak, Christopher W. Fletcher, Ling Ren, Nishanth Chandran, Satya V. Lokam, Elaine Shi, and Vipal Goyal. 2017. HOP: Hardware makes obfuscation practical. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium (NDSS'17)*.

[41] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious multi-party machine learning on trusted processors. In *Proceedings of the 25th USENIX Security Symposium (USENIX Security'16)*.

[42] Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of the 17th International Conference on Theory and Application of Cryptographic Techniques (EUROCRYPT'99)*.

[43] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: Protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*.

[44] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2015. Raccoon: Closing digital side-channels through obfuscated execution. In *Proceedings of the 24th USENIX Security Symposium (USENIX Security'15)*.

[45] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *Proceedings of the 14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*.

[46] Nigel Smart. 2014. *Algorithms, Key Size and Parameters Report*. ENISA.

[47] Geoffrey Smith. 2007. Principles of secure information flow analysis. In *Malware Detection*, Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang (Eds.). Advances in Information Security, Vol. 27. Springer, 291–307. https://doi.org/10.1007/978-0-387-44599-1_13

[48] Dawn Xiaoding Song, D. Wagner, and A. Perrig. 2000. Practical techniques for searches on encrypted data. In *Proceedings of the 2000 Symposium on Security and Privacy (S&P'00)*.

[49] Sai Tetali, Mohsen Lesani, Rupak Majumdar, and Todd Millstein. 2013. MrCrypt: Static analysis for secure cloud computations. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'13)*.

[50] Shruti Tople, Shweta Shinde, Zhaofeng Chen, and Prateek Saxena. 2013. AUTOCRYPT: Enabling homomorphic computation on servers to protect sensitive web content. In *Proceedings of the ACM International Conference on Computer and Communications Security (CCS'13)*.

[51] Daniel Wasserrab, Denis Lohner, and Gregor Snelting. 2009. On PDG-based noninterference and its modular proof. In *Proceedings of the 2009 Workshop on Programming Languages and Analysis for Security (PLAS'09)*.

Received April 2021; revised October 2021; accepted January 2022