Andreas Fischer*, Benny Fuhry, Florian Kerschbaum, and Eric Bodden

Computation on Encrypted Data using Dataflow Authentication

Abstract: Encrypting data before sending it to the cloud protects it against attackers, but requires the cloud to compute on encrypted data. Trusted modules, such as SGX enclaves, promise to provide a secure environment in which data can be decrypted and then processed. However, vulnerabilities in the executed program, which becomes part of the trusted code base (TCB), give attackers ample opportunity to execute arbitrary code inside the enclave. This code can modify the dataflow of the program and leak secrets via SGX side-channels. Since any larger code base is rife with vulnerabilities, it is not a good idea to outsource entire programs to SGX enclaves. A secure alternative relying solely on cryptography would be fully homomorphic encryption. However, due to its high computational complexity it is unlikely to be adopted in the near future. 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 above. We implemented DFAuth using a novel authenticated homomorphic encryption scheme, a Java bytecode-to-bytecode compiler producing fully executable programs, and an SGX enclave running a small and program-independent TCB. We applied DFAuth to an existing neural network that performs machine learning on sensitive medical data. The transformation yields a neural network with encrypted weights, which can be evaluated on encrypted inputs in 0.86 s.

Keywords: Dataflow Authentication, Trusted Code Base, Homomorphic Encryption, Authenticated Encryption, Secure Cloud Computing

DOI 10.2478/popets-2020-0002
Received 2019-05-31; revised 2019-09-15; accepted 2019-09-16.

***Corresponding Author: Andreas Fischer:** SAP Security Research, Karlsruhe, Germany, E-mail: andreas.fischer02@sap.com

Benny Fuhry: SAP Security Research, Karlsruhe, Germany, E-mail: benny.fuhry@sap.com

Florian Kerschbaum: School of Computer Science, University of Waterloo, Canada, E-mail: fkerschbaum@uwaterloo.ca

Eric Bodden: Heinz Nixdorf Institute, University of Paderborn, Germany: eric.bodden@upb.de

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 compute on encrypted data.

Trusted modules, e.g. secure enclaves such as Intel SGX [1–3], 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 an attacker ample opportunity to execute arbitrary code in the enclave [4]. 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 [5–7]. Since the number of software vulnerabilities scales 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 a good idea to outsource entire programs to an SGX enclave. Our approach significantly reduces the surface for such attacks by providing a small, program-independent TCB.

Approaches obfuscating the control flow [8–11] cannot prevent exploiting vulnerabilities in the Trusted Module, but could be applied to the unprotected program in our approach to strengthen it further. For example, HOP [10] obfuscates programs by encrypting them such that only a trusted processor can decrypt and run them. However, HOP assumes that the program is free of software vulnerabilities and runs the entire program inside the Trusted Module. In contrast, in our approach the Trusted Module is small and program-independent

and application vulnerabilities are confined to the untrusted program computing on encrypted data.

Consider the following dataflow modification attack that efficiently leaks a secret x in its entirety. We 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; the "else" branch otherwise. This can be observed, for example, by the branch shadowing attack presented in [6]. 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 encrypted 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 defence 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), such that only variables with a pre-approved 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 we mentioned before. 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 which 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 [12] would be another alternative to compute on encrypted data without the drawback of data-leaks. Due to its high computational complexity [13], however, researchers are seeking efficient alternatives that offer similar security. Fortunately, we know how to efficiently perform additively and multiplicatively homomorphic operations on 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 [14], JCrypt [15] and AutoCrypt [16] 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.

Table 1. Comparison of DFAuth to the most relevant alternative approaches computing on encrypted data. Note that AutoCrypt only supports control flow decisions on input variables.

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

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 modules are small and program-independent and are run protected in the SGX enclave whereas the program runs efficiently on homomorphic encrypted values in unprotected memory. 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 authenticated homomorphic encryption scheme HASE.

For a summary of key properties provided by DFAuth and a comparison to the most relevant alternative approaches computing on encrypted data, we refer the reader to Table 1.

We implemented the program transformation in a bytecode-to-bytecode compiler, such that the resulting programs are executable. We evaluate DFAuth based on two applications: a checkout (shopping cart) component of a sales application and a neural network performing evaluations on sensitive medical data. The transformed applications execute in 2.3 ms and 0.86 s, respectively. This shows that DFAuth is practically deployable, whilst also providing extensive security guarantees.

In summary, our contributions are:

- We define the concept of *dataflow authentication* and show its interference equivalence property in a program dependency graph.
- We present a new *authenticated homomorphic encryption scheme* HASE, which can be used for constant time implementation of dataflow authentication.

– We implemented and evaluated a *bytecode-to-bytecode program transformation* for computation on encrypted data using dataflow authentication.
– We implemented and evaluated transformed programs, e.g., machine learning, using *Intel SGX*.

Structure of this Work

This paper is structured as follows. In the next section, we provide our adversary model and various definitions of our authenticated homomorphic encryption scheme HASE. In Section 3, we introduce dataflow authentication DFAuth and the security it provides. Section 4 presents our HASE constructions and discusses their security. Details about our implementation in Java are given in Section 5 and Section 6 shows the results of our evaluation using this implementation. Section 7 presents related work before Section 8 concludes our work.

2 Definitions

In order to understand the security of dataflow authentication, we first define the overall adversary model considered, the algorithms that HASE offers and the security it guarantees.

2.1 Adversary Model

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. Figure 1 depicts the process and its trust boundaries. The client wishes to execute a program at the cloud server with sensitive input data. 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 distinguish two phases of this outsourced computation: setup and runtime. First, the client chooses the keys for the encryption of its inputs in our HASE scheme (A). Then the client transforms the intended program using a specialized HASE-enabled compiler (B) and uploads it to the cloud. The server deploys some parts of the program into the Trusted Module which 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

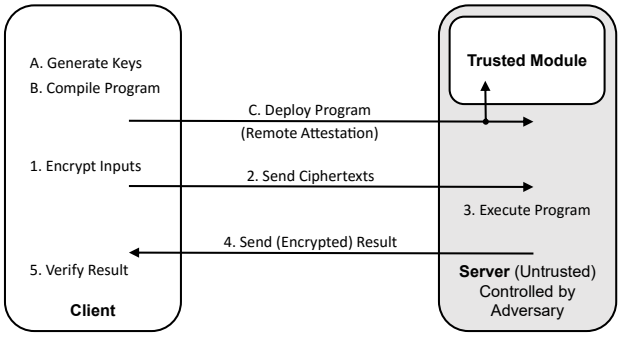

Fig. 1. System Overview

the compiled program and sends the ciphertexts to the cloud server (1-2). The cloud server now executes the program (3). We assume an active adversary controlling the cloud. The adversary can

– *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, e.g., by breaking the program, executing instructions step-by-step and modifying the instruction pointer (except in the Trusted Module).
– do all of this arbitrarily *interleaved*.

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).

We ensure the following security property: The server has *learnt nothing beyond the intended information flow* of the program to unclassified memory locations (interference equivalence).

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 [9] or both branches of a conditional can be executed and the result combined using an oblivious store operation [11].

2.2 Notation

We denote an object whose members can be unambiguously accessed individually as $\langle \ldots \rangle$. 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 com-

parisons. 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 $\pi_i(S) := \{\pi_i(x) : x \in S\}$.

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. 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 $\cdot$ denoting that $\mathcal{A}$ may freely choose this parameter.

2.3 Game-Based Security

We provide security definitions as *games* (security experiments) played between a PPT *challenger* and a PPT *adversary* $\mathcal{A}$ [17]. 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 no adversary has non-negligible advantage. The proof is achieved by reducing the winning of the game to some problem that is assumed to be hard.

2.4 Homomorphic Authenticated Symmetric Encryption (HASE)

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 [18]. Unforgeability of HASE schemes (HASE-UF-CPA) is based on the common *unforgeable encryption* definition [18].

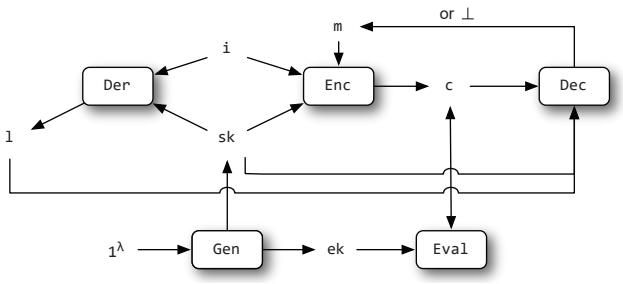

Fig. 2. HASE Overview

Definition 1 (HASE Syntax)**.** *A HASE scheme is a tuple of PPT algorithms* $\langle \mathsf{Gen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Der}, \mathsf{Dec} \rangle$ *such that:*

- *The* key-generation algorithm Gen *takes the security parameter* 1^λ *as input and outputs a key pair* $\langle ek, sk \rangle$ *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* $(\mathcal{C}, \otimes)$ *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* $c \in \mathcal{C}$*.*
- *The* evaluation algorithm Eval *takes an evaluation key ek and a set of ciphertexts* $C \subseteq \mathcal{C}$ *as input and outputs a ciphertext* $\hat{c} \in \mathcal{C}$*.*
- *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* $c \in \mathcal{C}$ *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* Π *be a HASE scheme consisting of five algorithms as described above. Furthermore, let* $S := \{(m, i) : m \in \mathcal{M}, i \in \mathcal{I}\}$ *be a set of plaintexts with unique identifiers and let* $I := \pi_2(S)$ *be the set of identifiers in* S*. We say that* Π *is* correct *if for any honestly generated key pair* $\langle ek, sk \rangle \leftarrow \Pi.\mathsf{Gen}(1^\lambda)$ *and any set of ciphertexts* $C := \{c : c \leftarrow \Pi.\mathsf{Enc}(sk, m, i) : (m, i) \in S\}$ *it holds that*

$$\Pi.\mathsf{Dec}(sk, \Pi.\mathsf{Eval}(ek, C), \Pi.\mathsf{Der}(sk, I)) = \bigoplus_{(m,i) \in S} m$$

except with negligible probability over $\langle ek, sk \rangle$ *output by* $\Pi.\mathsf{Gen}()$ *and all c output by* $\Pi.\mathsf{Enc}()$*.*

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* $\mathsf{negl}(\lambda)$ *such that*

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

The experiment is defined as follows:

$\underline{\text{ExpHASE}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(\lambda)}$

$S := \{\}$
$\langle ek, sk\rangle \leftarrow \Pi.\mathsf{Gen}(1^\lambda)$
$\langle m_0, m_1, i, \mathsf{st}\rangle \leftarrow \mathcal{A}^{E_{sk}}(1^\lambda, ek)$
if $i \in \pi_2(S)$ **then**
  return 0
else
  $b \leftarrow\!\!\$\ \{0,1\}$
  $S := S \cup \{(m_b, i)\}$
  $c \leftarrow \Pi.\mathsf{Enc}(sk, m_b, i)$
  $b' \leftarrow \mathcal{A}^{E_{sk}}(1^\lambda, c, \mathsf{st})$
  return $b = b'$

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

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

We incorporate the identifier parameter to the encryption algorithm by allowing the adversary to submit an additional identifier argument to encryption oracle queries. The oracle enforces uniqueness of the identifier by keeping track of all queried plaintext-identifier pairs. It rejects any encryption query involving a previously used identifier. Furthermore, the adversary is allowed to pick the identifier used for the encryption of the challenge plaintext m_b.

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* $\mathsf{negl}(\lambda)$ *such that*

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

with the experiment defined as follows:

$\underline{\text{ExpHASE}^{\text{UF-CPA}}_{\mathcal{A},\Pi}(\lambda)}$

$S := \{\}$
$\langle ek, sk\rangle \leftarrow \Pi.\mathsf{Gen}(1^\lambda)$
$\langle c, I\rangle \leftarrow \mathcal{A}^{E_{sk}}(1^\lambda, ek)$
$l := \Pi.\mathsf{Der}(sk, I)$
$m := \Pi.\mathsf{Dec}(sk, c, l)$
$\tilde{m} := \bigoplus_{(m',i)\in S, i\in I} m'$
return $m \neq \perp \wedge m \neq \tilde{m}$

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

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

The adversary returns a ciphertext c and a set of identifiers I. The adversary is successful if and only if two conditions are met. First, c has to successfully decrypt under the label derived from I. Second, the resulting plaintext m must be 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 an example. Consider the following excerpt from a Java program:

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

First DFAuth performs a conversion to single static assignment (SSA) form [19]: assign each variable at exactly one code location; create atomic expressions; introduce fresh variables if required. In the example, DFAuth changes the program to the following:

```
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);
```

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 [16]. 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 appropriate type. HASE implements multiplicative homomorphic encryption *mul* and its operations directly, while 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:

Listing 1. Example as executed on the server

```
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);
```

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 4 in Section 5 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, since 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 paper, we thus propose to restrict the dataflow using DFAuth. This allows such conversions in a secure way by enforcing that encrypted variables can be decrypted only along the program's original dataflow. The approach comprises two steps. First, happening at compile time, for each conversion DFAuth pre-computes 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 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 $\text{ExpHASE}^{\text{IND-CPA}}_{\mathcal{A},\Pi}$. This modification to binary search can only succeed if the decryption operations Dec in `convertToCmpGT42` (or other conversion routines) succeed. The adversary can minimize the Dec operations, e.g., by not introducing new calls to conversion routines, but given the scheme defined above, any attempt to alter the dataflow on encrypted variables will cause Dec to fail: Assume that an attacker inserts code in Listing 1 to search for the secret value `d` (resulting code shown in Listing 2). 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 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`.

Listing 2. Example modified by the attacker

```
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  }
```

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* [20], 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. While 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 Program Dependency Graphs (PDGs) that capture the program's control and data flow [21]. 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 paper one must assume that the executed program *before encryption* already shows interference for some memory locations, e.g., 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. Then even in the setting here 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 paper 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, e.g., the transformation explained above, which inserted a binary search. Then the property we would like to obtain is that:

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

In other words: disregarding the explicitly declassified information within C, the transformed program does not leak any additional information, i.e., the adversary cannot learn any additional information about the encrypted data. Let us assume for a moment that the above 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, which albeit not being 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 [22]. 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.

Construction 1 (Multiplicative HASE)**.** *Let $\mathcal{G}$ be a group generation algorithm (cf. Definition 6). Define a HASE scheme using the following PPT algorithms:*

Gen*: on input 1^λ obtain $\langle \mathbb{G}, q, g\rangle \leftarrow \mathcal{G}(1^\lambda)$. For a pseudorandom function 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 $\mathbb{G}$, the secret key is $\langle \mathbb{G}, q, g, a, x, y, h, j, k\rangle$. 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 $sk = \langle \mathbb{G}, q, g, a, x, y, h, j, k\rangle$, 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 $\langle u, v, w\rangle$.*

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 $\langle \mathbb{G}, q, g, a, x, y, h, j, k\rangle$ 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 $\langle \mathbb{G}, q, g, a, x, y, h, j, k\rangle$, a ciphertext $c = \langle u, v, w\rangle$ 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 $\mathbb{G}$. As can be easily seen, 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 $\mathbb{G}_q$ on the other hand 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. [23] to Construction 1. The basic idea 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 $\langle g^r, h^r g^m\rangle$ rather than $\langle g^r, h^r m\rangle$. 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:

$$\langle g^{r_1} \cdot g^{r_2}, h^{r_1} g^{m_1} \cdot h^{r_2} g^{m_2}\rangle = \left\langle g^{r_1+r_2}, h^{r_1+r_2} \cdot g^{m_1+m_2}\right\rangle$$

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 6) to be hard relative to $\mathcal{G}$. Hu et al. keep exponents small enough for discrete logarithm algorithms to terminate within reasonable time despite their exponential asymptotic runtime. 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 [24].

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 $\langle \mathbb{G}, q, g\rangle \leftarrow \mathcal{G}(1^\lambda)$. For a pseudorandom function family $H : \mathcal{K} \times \mathcal{I} \to \mathbb{Z}_q$ 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 $D := \langle d_1, \ldots, d_t, d\rangle$. Choose $a, x, y \leftarrow_\$ \mathbb{Z}_q$ and compute $h := g^x$, $j := g^y$. The evaluation key is $\mathbb{G}$, the secret key is $\langle \mathbb{G}, q, g, a, x, y, h, j, k, D\rangle$. 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 $sk = \langle \mathbb{G}, q, g, a, x, y, h, j, k, D\rangle$, 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 $\langle u_1, v_1, \ldots, u_t, v_t, s, w\rangle$.

Der*: on input a secret key $\langle \mathbb{G}, q, g, a, x, y, h, j, k, D\rangle$ 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 $\langle \mathbb{G}, q, g, a, x, y, h, j, k, D\rangle$, a ciphertext $\langle u_1, v_1, \ldots, u_t, v_t, s, w\rangle$ and a secret label $l \in \mathbb{G}$. Parse $D = \langle d_1, \ldots, d_t, d\rangle$. 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 5 (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 pseudorandom function (PRF), if for all PPT algorithms $\mathcal{A}$ there is a negligible function $\mathsf{negl}(\lambda)$ such that:*

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

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

Definition 6 (DDH Problem [18])**.** *Let $\mathcal{G}$ be a PPT algorithm taking 1^λ as input and outputting $\langle \mathbb{G}, q, g\rangle$ where $\mathbb{G}$ is a description of a cyclic group, q is the order of $\mathbb{G}$ and g is a generator of $\mathbb{G}$. 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:*

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

where in each case the probabilities are taken over the experiment in which $\mathcal{G}(1^\lambda)$ outputs $\langle \mathbb{G}, q, g\rangle$, 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 Implementation

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

5.1 Setup Phase

The setup phase is divided into two parts, *compilation* and *deployment*. An overview is provided in Figure 3.

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 HASE 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 [25].

Our implementation uses a crypto library to encrypt program constants and choose variable labels (2-3). The crypto library contains implementations of all required cryptographic algorithms, including our own from Section 4. It implements the PRF used for the authentication labels as HMAC-SHA256 [26]. For the group operations in Multiplicative HASE we use MPIR [27] for large integer arithmetic. Additive HASE operates on the elliptic curve group provided by `libsodium` [28]. 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 using exhaustive search with a fixed set of precomputed values.

Our compiler converts floating-point plaintexts to a fixed-point representation by an application-defined scaling factor. It also transforms the calculations to integer values, whereby the scaling factors are considered when appropriate. To ensure the efficiency of Additive HASE decryption, the compiler can insert Trusted Module invocations into the program that decrypt and re-encrypt Additive HASE 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 also be defined by the application. We demonstrate the efficacy of these re-encryptions in Appendix B. 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. SGX is well suited for our implementation, because it provides remote attestation, secure data storage and isolated program execution. It is available in Intel Core processors beginning at the Skylake generation and in some Intel Xeon processors, for example the Kaby Lake generation. Using the remote attestation feature of SGX the client prepares the enclave (refer to [1] for details). This feature allows to verify the correct creation of an enclave in a remote system and – in our case – the correct setup of the crypto library. Additionally, SGX's remote attestation provides means to establish a secure channel between an external party and an enclave, over which we transfer the sensitive conversion data to the untrusted cloud server. We emphasize that cryptographic keys and conversion data is protected from access by any software except the enclave by SGX's hardware protection.

5.2 Runtime Phase

To run the program, the client executes the main class which triggers the remote program execution at the cloud server (see Figure 4). 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. It forwards re-encryption and comparison requests to the Trusted Module and

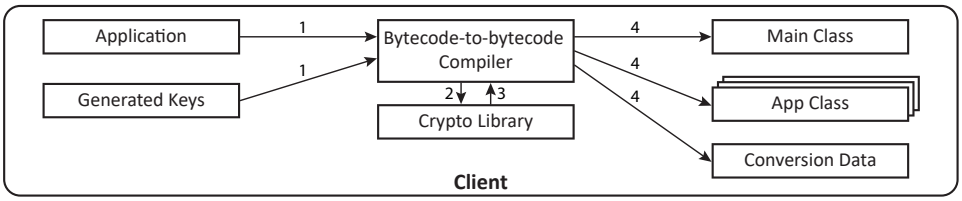

Fig. 3. Application Transformation during setup phase. See Section 5.1 for a full description.

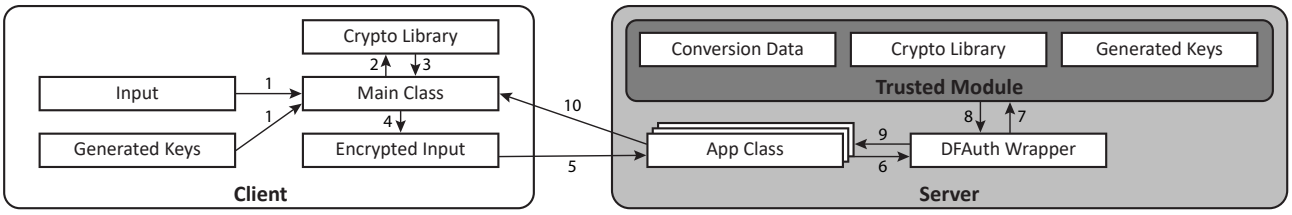

Fig. 4. Application Execution during Runtime Phase. See Section 5.2 for a full description.

passes the answers back to the application (7-9). Once the app classes have finished 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.

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 homomorphic encryption and comparison in Listings 3 and 4. The conversion routine to additively homomorphic encryption is similar to the one for multiplicative encryption in Listing 3 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 4. 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.

Listing 3. Conversion to multiplicative HE

```
1  convertToMul(x, "x") {
2    label = labelLookup("x");
3    y = Dec(K, x, label);
4    if (y == fail)
5      stop;
6    id = idLookup("x");
7    return Enc(K, y, id);
8  }
```

Listing 4. Conversion to comparison

```
1  convertToCmp(x, y, "x") {
2    label = labelLookup("x");
3    x1 = Dec(K, x, label);
4    if (x1 == fail)
5      stop;
6    if (y == null) {
7      param = paramLookup("x");
8      switch (param.type) {
9        case EQ:
10         return (x1 == param.const);
11       case GT:
12         return (x1 > param.const);
13       case GTE:
14     ...
15   } else {
16     label = labelLookup("y");
17     y1 = Dec(K, y, label);
18     if (y1 == fail)
19       stop;
20     ...
21   }
22 }
```

6 Evaluation

In this section, we present the evaluation results collected in two experiments. In our first experiment, we apply DFAuth to the checkout (shopping cart) component of a *Secure Sales Application*, which we developed ourselves. In our second experiment, we use DFAuth to transform an existing neural network program enabling *Secure Neural Networks in the Cloud*.

All experiments were performed on an Intel Core i7-6700 CPU with 64 GB RAM running Windows 10. For the evaluation, we aimed for a security level equivalent to 80 bits of symmetric encryption security. We used the 1536-*bit MODP Group* from RFC3526 [29] as the underlying group in Multiplicative HASE. The `libsodium` [28] elliptic curve group used by Additive HASE even provides a security level of 128 bits [30].

6.1 Secure Sales Application

In this experiment we consider the checkout component of a secure sales application running on an untrusted server. When a client checks out a shopping cart, the server is tasked with summing up the encrypted prices of all items in the cart. Additionally, discounts need to be applied when the sum exceeds certain thresholds. We aim to protect the rebate structure of the client, i.e. the thresholds when discounts are applied and the percentage of the discount. This is important in outsourced sales applications, because an attacker, e.g. a co-residing competitor, could try to infer the rebate structure in order to gain an unfair advantage in the market.

The purpose of this experiment is twofold. First, this experiment serves as a performance test for the implementation of our HASE constructions. Second, this experiment serves as an example of the larger class of non-linear functions. While the secrets in this illustrative example can be inferred fairly easily, with growing number of parameters inference gets harder. Note that the constants in almost any computation can be inferred by repetitive queries. It is a question of costs whether it is economically worthwhile to do it. Also note that in a business to business sales application, almost each customer has its own confidential rebate structure.

The transformed program performs control flow decisions comparing the sum of encrypted inputs to the encrypted thresholds of the rebate structure. To a control flow observer each such decision reveals a small amount of information about the relationship between inputs and threshold constants. After observing the result of sufficiently many such decisions, an attacker may be capable of inferring the confidential threshold values. Note that the attacker does not learn addition information about the output and most importantly the discounts granted. If all leakage is to be prevented, the techniques described in Section 2.1 can be applied additionally.

Experimental Setup.
We implemented this checkout component and applied DFAuth to it. If the sum exceeds the value of \$250, our implementation grants a discount of 5%. If the sum exceeds the value of \$500, we grant a total discount of 10%.

In order to evaluate the performance of the original (*plaintext*) and the DFAuth variant of the program, we built shopping carts of sizes $\{1, 10, \ldots, 100\}$. Prices were taken uniformly at random from the interval $[0.01, 1000.00]$. For each cart a plaintext and an encrypted variant is stored at the untrusted server.

For each of the two program variants, the total runtime of code executing at the untrusted server was measured. This time includes reading the corresponding cart data from disk, summing up the prices of all cart items and granting discounts where applicable. For the DFAuth variant, we also collected the number of operations performed on encrypted data inside and outside of the Trusted Module, as well as the time spent invoking and inside the Trusted Module. Our measurements do not include the setup phase, because it is only a one-time overhead that amortizes over multiple runs. We also do not include network latency in our measurements, since the difference in communication between a program running on plaintext and a program running on encrypted data is very small.

Evaluation Results.
There are three cases for the control flow in the secure sales application:

1. The sum of all item prices neither reaches the first threshold nor the second threshold. In this case, the sum of the prices is compared to two different threshold constants.
2. The sum of all prices reaches the larger threshold. In this case, the sum is compared to one threshold constant and needs to be converted to Multiplicative HASE before being multiplied with the respective discount constant.

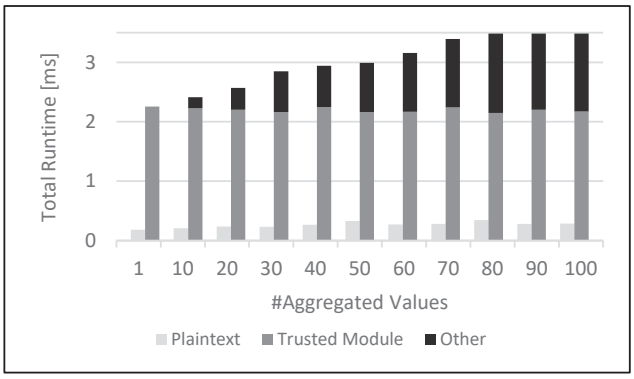

Fig. 5. Mean runtime [ms] of the original (left) and DFAuth (right) variants of the shopping cart program as a function of the cart size for Case 3. We do not show the 95% confidence interval, because even the largest value is only $\pm$ 0.003 ms, which would not be visible in the graph.

3. The sum of all prices reaches the lower threshold, but not the larger threshold. In this case, the sum is compared to two threshold constants and needs to be converted to Multiplicative HASE before being multiplied with the respective discount constant.

Figure 5 presents the average runtime of 100 runs of the experiment described above for Case 3. We can see that even for large-sized carts, containing 100 items, DFAuth only increases the program runtime by a factor of 11.5 on average. Assuming a round-trip latency of 33 ms, which can be considered realistic according to [31], the total round-trip time (consisting of round-trip latency and program runtime) increases only by a factor of 1.08 on average. Most importantly, the absolute runtime values are sufficiently low for practical deployment of online computation. From Figure 5 we can also see that 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, which reports the number of operations performed on encrypted data inside and outside of the Trusted Module, shows that this is not the case. Even for a shopping cart containing only 10 items, there are 9 to 10 untrusted HASE operations, but only 1 to 3 trusted operations. While the number of trusted operations is independent of the cart size, the number of untrusted operations is approximately linear in the cart size, i.e., an even larger cart would further increase the fraction of untrusted operations. We refer to Appendix B for a demonstration of Additive HASE's scalability.

Table 2. Number of untrusted (HASE) and trusted (SGX) operations in the Secure Sales Application experiment for shopping cart size 10. [Notation: Case 1/2/3]

Operation	Type	# Ops
Homomorphic Addition	HASE	9/9/9
Homomorphic Multiplication	HASE	0/1/1
Total	HASE	9/10/10
Additive to Multiplicative Conversion	SGX	0/1/1
Comparison to Constant	SGX	2/1/2
Total	SGX	2/2/3

6.2 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, i.e., 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. Also, 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. Also, 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 [32], a framework for neural networks. Given a set of features extracted from an image of a breast tumour, the network predicts whether the tumour 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 data set 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, the test set is used to evaluate whether the network delivers correct predictions.

We start by applying our DFAuth mechanism 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 using HASE. 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. Inputs are sampled uniformly at random (without replacement) from the test data set. Again, we measure the total runtime 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. The total runtime includes reading the network configuration (i.e., layers and neuron), loading the weights and executing the evaluation.

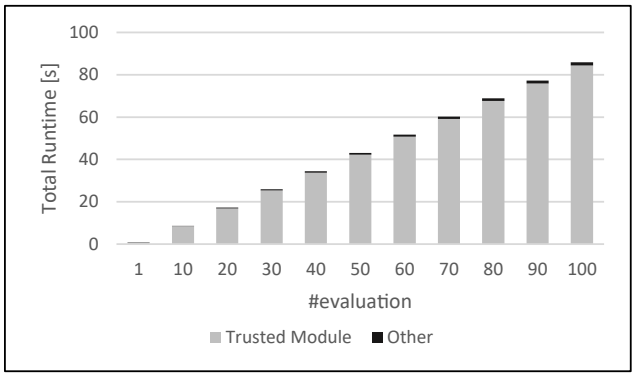

Fig. 6. Mean runtime [s] of the DFAuth variant of the breast cancer neural network experiment as a function of the number of evaluations. We do not show the 95% confidence interval, because even the largest value is only $\pm$ 0.02 s, which would not be visible in the graph. For the same reason, we do not include the plaintext measurements.

Table 3. Number of untrusted (HASE) and trusted (SGX) operations for a single evaluation of the neural network.

Operation	Type	# Ops
Homomorphic Addition	HASE	548
Homomorphic Multiplication	HASE	548
Total	HASE	1096
Additive to Multiplicative Conversion	SGX	36
Multiplicative to Additive Conversion	SGX	548
Comparison to Constant	SGX	36
Total	SGX	620

Evaluation Results.
We present the evaluation results of the encrypted neural network in Figure 6. The total runtime of one network evaluation is only 0.86 s, whereby 0.84 s (98%) are spend in the Trusted Module (SGX) and 0.02 s (2%) outside of the Trusted Module. Even for 100 evaluations the run completes in 85.96 s on average. In this case, the processing time in the Trusted Module is 84.39 s (98%) and 1.56 s (2%) outside. The relative runtime of an evaluation (total runtime / number of network evaluations) is 860 ms with a 95% confidence interval of $\pm$ 0.42 ms. This waiting time of less than one second should be acceptable for a user, and demonstrates the practical deployment of neural network evaluation on encrypted data. Compared to one plaintext network evaluation, the runtime increased by a factor of about 677.

As in the previous experiment, a large portion of the total runtime is spend inside the Trusted Module. Table 3 reports the number of untrusted operations and number of Trusted Module operations. For a single neural network evaluation, 1096 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.

Comparison to Alternative Solutions.
Recently, implementations of machine learning on encrypted data have been presented for somewhat homomorphic encryption [33] and Intel SGX [34]. Compared to the implementation on somewhat homomorphic encryption our approach offers the following advantages: First, our approach has a *latency* of 0.86 seconds compared to 570 seconds for somewhat homomorphic encryption. The implementation in [33] exploits the inher-

ent 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 [33] 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 [33] due to computation on integers does not apply in our case. However, we have not evaluated this aspect of accuracy in comparison to [33]. Finally, our approach is capable of *outsourcing* a neural network evaluation whereas the approach in [33] 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 runtime overhead is smaller than ours, our approach offers the following advantage compared to the implementation on SGX [34]: In our approach the code in the SGX enclave is *independent of the functionality*, e.g., machine learning. The implementation in [34] provides a new, specific algorithm for each additional machine learning function, i.e., neural networks, decision trees, etc. Each implementation has been specifically optimized to avoid side channels on SGX and hopefully scrutinized for software vulnerabilities. The 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 Related Work

Our work is related to obfuscation techniques and trusted hardware, (homomorphic) authenticated encryption and computation over 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. Molnar et al. [9] eliminate control-flow side-channels by transforming code containing conditional instructions into straight-line code employing masking. GhostRider [8] 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 [11] 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 [10] 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 (AE) is an encryption mode that provides confidentiality as well as authenticity (unforgeability). An AE can be obtained by composing an IND-CPA secure encryption scheme with a signature or message authentication code (MAC) [35]. Hence, one can obtain a *homomorphic* AE by combining a homomorphic encryption scheme with a homomorphic MAC. However, since the best known homomorphic MACs [36] are not yet fully homomorphic a different construction is required. Joo and Yun provide the first fully homomorphic AE [37]. However, their decryption algorithm is as complex as the evaluation on ciphertexts undermining the advantages of an encrypted program, i.e., one could do the entire computation in the Trusted Module. In parallel work, Barbosa et al. develop labeled homomorphic encryption [38] which, however, has not been applied to Trusted Modules.

Boneh et al. [39] 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. [40] integrated MACs into efficient, linearly homomorphic Paillier encryption [41] and used identifiers to support public verifiability. 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 pre-computed derivation phase and a verification phase. Hence, we can achieve constant time verification.

Aggregate MACs [42] provide support for aggregation of MACs from distinct keys. However, our current dataflow authentication considers one client and secret key.

Computation over Encrypted Data.
Since fully homomorphic encryption [12] entails rather high computational overhead [13] researchers have resorted to partially encrypting computations. MrCrypt [14] 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. However, still no conversions between encryption schemes were performed. AutoCrypt [16] used these conversions, however, 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. used deterministic encryption to implement a large subset of SQL [43]. Popa et al. used also randomized and order-preserving encryption in an adjustable manner [44].

Verifiable computation [45] 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 paper. It only proves that the server performed one correct computation, but not that it did not perform any others.

Functional encryption [46] is a more powerful computation on encrypted data than homomorphic encryption. It not only can compute any function, but also reveal the result of the computation and not only its ciphertext. However, generic constructions [47] 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.

8 Conclusions

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 in order 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 a novel authenticated, homomorphic encryption scheme and Trusted Modules in an SGX enclave. Compared to an implementation solely on fully homomorphic encryption we offer better and actually practical performance and compared to an implementation solely on Intel's SGX we offer a much smaller trusted code base 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.

9 Acknowledgements

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

References

[1] Ittai Anati, Shay Gueron, Simon P. Johnson, and Vincent R. Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Workshop on Hardware and Architectural Support for Security and Privacy*, HASP, 2013.

[2] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using Innovative Instructions to Create Trustworthy Software Solutions. In *Workshop on Hardware and Architectural Support for Security and Privacy*, HASP, 2013.

[3] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Workshop on Hardware and Architectural Support for Security and Privacy*, HASP, 2013.

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

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

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

[7] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 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, 2017.

[8] Chang Liu, Austin Harris, Martin Maas, Michael W. Hicks, Mohit Tiwari, and Elaine Shi. 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, 2015.

[9] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*, ICISC, 2005.

[10] Kartik Nayak, Christopher W. Fletcher, Ling Ren, Nishanth Chandran, Satya V. Lokam, Elaine Shi, and Vipal Goyal. HOP: hardware makes obfuscation practical. In *24th Annual Network and Distributed System Security Symposium*, NDSS, 2017.

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

[12] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Symposium on Theory of Computing*, STOC, 2009.

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

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

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

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

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

[18] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography, Second Edition*. Chapman & Hall/CRC, 2nd edition, 2014.

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

[20] Geoffrey Smith. Principles of secure information flow analysis. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 291–307. Springer, 2007.

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

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

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

[24] Nigel Smart. Algorithms, key size and parameters report, 2014.

[25] Patrick Lam, Eric Bodden, Ondrej Lhotak, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop*, CETUS, 2011.

[26] D. Eastlake 3rd and T. Hansen. US Secure Hash Algorithms (SHA and HMAC-SHA). RFC 4634 (Informational), 2006.

[27] Mpir: Multiple precision integers and rationals. http://mpir.org.

[28] The sodium crypto library (libsodium). https://download.libsodium.org/doc/.

[29] T. Kivinen and M. Kojo. More modular exponential (modp) diffie-hellman groups for internet key exchange (ike). RFC 3526 (Proposed Standard), 2003.

[30] Daniel J. Bernstein. Curve25519: New diffie-hellman speed records. In *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*.

[31] At&t gloabl ip network - network averages. http://ipnetwork.bgtmo.ip.att.net/pws/averages.html.

[32] Neuroph – java neural network framework. http://neuroph.sourceforge.net.

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

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

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

[36] Rosario Gennaro and Daniel Wichs. Fully homomorphic message authenticators. In *Proceedings of the 19th International Conference on the Advances in Cryptology*, ASIACRYPT, 2013.

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

[38] Manuel Barbosa, Dario Catalano, and Dario Fiore. Labeled homomorphic encryption - scalable and privacy-preserving processing of outsourced data. In *Proceedings of the 22nd European Symposium on Research in Computer Security*, ESORICS, 2017.

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

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

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

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

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

[44] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP, 2011.

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

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

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

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

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 $\mathsf{Adv}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(\lambda)$ 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 7.

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

Game 1 (Indistinguishability-Based Transition).
Instead of deriving the label used in the third component of the challenge ciphertext using the pseudorandom function $H : \mathcal{K} \times \mathcal{I} \to \mathbb{G}$ 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 pseudorandom function, 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 and f is a random function, the result is a random element of $\mathcal{G}$. Thus, instead of computing $l := f(i)$, 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

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

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

$S := \{\}$
$\langle \mathbb{G}, q, g\rangle \leftarrow \mathcal{G}(1^\lambda)$
$a, x, y \leftarrow\!\!\$\, \mathbb{Z}_q, k \leftarrow\!\!\$\, \mathcal{K}$
$h := g^x, j := g^y$
$ek := \mathbb{G}$
$sk := \langle \mathbb{G}, q, g, a, x, y, h, j, k\rangle$
$\langle m_0, m_1, i, \mathsf{st}\rangle \leftarrow \mathcal{A}^{E_{sk}}(1^\lambda, ek)$
if $i \in \pi_2(S)$ **then**
  return 0
else
  $b \leftarrow\!\!\$\, \{0,1\}$
  $S := S \cup \{(m_b, i)\}$
  $r, \boxed{s}, \boxed{z} \leftarrow\!\!\$\, \mathbb{Z}_q, l := \boxed{g^s}$
  $c := \left\langle g^r, \boxed{g^z} \cdot m_b, j^r \cdot {m_b}^a \cdot l \right\rangle$
  $b' \leftarrow \mathcal{A}^{E_{sk}}(1^\lambda, c, \mathsf{st})$
  return $b = b'$

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

parse $sk = \langle \mathbb{G}, q, g, a, x, y, h, j, k\rangle$
if $i \in \pi_2(S)$ **then**
  return $\perp$
else
  $S := S \cup \{(m, i)\}$
  $r \leftarrow\!\!\$\, \mathbb{Z}_q, l := H(k, i)$
  $c := \langle g^r, h^r \cdot m, j^r \cdot m^a \cdot l\rangle$
  return c

Fig. 7. Final security experiment used in HASE-IND-CPA proof. Changes compared to the first experiment are highlighted.

In both cases $\mathcal{D}$ receives $\langle \mathbb{G}, q, g\rangle$ 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.

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 $\mathsf{Adv}^{\text{IND-CPA}}_{\mathcal{A},\Pi}(\lambda) = \mathsf{negl}(\lambda)$. □

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 $\mathsf{Adv}^{\text{UF-CPA}}_{\mathcal{A},\Pi}(\lambda)$ 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 8.

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

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 pseudorandom function $H(k,\cdot)$ 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 $\Pr[S_2] = \mathsf{negl}(\lambda)$. 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

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

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 $\langle c, I\rangle$ be the output of the adversary. The set of identifiers I determines a label l and an expected message $\tilde{m}$. Furthermore, let $\tilde{c} = \langle \tilde{u}, \tilde{v}, \tilde{w}\rangle$ be the ciphertext resulting from the application of $\Pi.\mathsf{Eval}$ 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, in order for $c = \langle u, v, w\rangle$ 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 $\tilde{w} \neq w$.

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

$S := \{\}$
$\langle \mathbb{G}, q, g\rangle \leftarrow \mathcal{G}(1^\lambda)$
$a, x, y \leftarrow\$ \mathbb{Z}_q, k \leftarrow\$ \mathcal{K}, \boxed{f \leftarrow\$ \mathcal{F}}$
$h := g^x, j := g^y$
$ek := \mathbb{G}$
$sk := \left\langle \mathbb{G}, q, g, a, x, y, h, j, \boxed{f} \right\rangle$
$\langle c, I\rangle \leftarrow \mathcal{A}^{E_{sk}}(1^\lambda, ek)$
$l := \prod_{i \in I} \boxed{f(i)}$
parse $c = \langle u, v, w\rangle$
$m := u^{-x} \cdot v$
$t := u^{y} \cdot m^{a} \cdot l$
$\tilde{m} := \bigoplus_{(m',i) \in S, i \in I} m'$
return $\boxed{t = w} \wedge m \neq \tilde{m}$

$E_{sk}(m, i)$

parse $sk = \left\langle \mathbb{G}, q, g, a, x, y, h, j, \boxed{f} \right\rangle$
if $i \in \pi_2(S)$ **then**
  return $\perp$
else
  $S := S \cup \{(m, i)\}$
  $r \leftarrow\$ \mathbb{Z}_q, l := \boxed{f(i)}$
  $c := \langle g^r, h^r \cdot m, j^r \cdot m^a \cdot l\rangle$
  return c

Fig. 8. Final security experiment used in HASE-UF-CPA proof. Changes compared to the first experiment are highlighted.

Combining equations (2) and (3) yields

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

In order 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).

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. [23]. 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, the other with re-encryptions performed after every 4000 homomorphic evaluations. We use $n \in \{10, 100, 1000, 10000, 100000\}$ and perform each measurement 100 times.

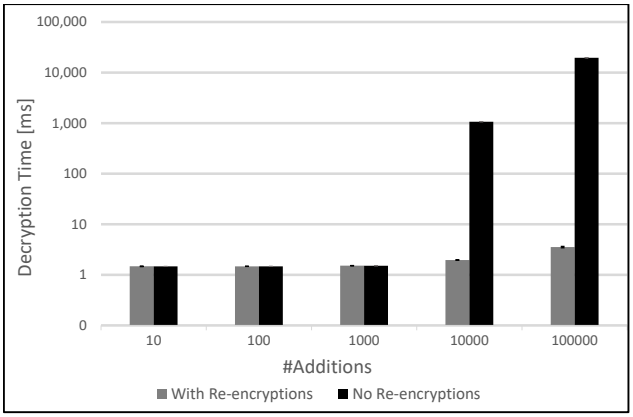

Fig. 9. Mean runtime [ms] to decrypt a ciphertext produced by summing up a varying number of ciphertexts using the Additive HASE scheme.

Evaluation Results.
Figure 9 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 1000 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 5.1). In comparison, when re-encryptions are performed, the decryption time only increases minimally, even for $n = 100000$.