Journal of Computer Security X (2017) 1–42
IOS Press

Assuring BetterTimes

Private arithmetic formulas

Per Hallgren [a], Ravi Kishore [b], Martín Ochoa [c,d], Andrei Sabelfeld [a]
[a] *Chalmers University of Technology, Gothenburg, Sweden*
[b] *International Institute of Information Technology, Hyderabad, India*
[c] *Singapore University of Technology and Design, Singapore*
[d] *Department of Applied Mathematics and Computer Science, Universidad del Rosario, Bogotá, Colombia*

Abstract.
We present a privacy-assured multiplication protocol using which an arbitrary arithmetic formula with inputs from two parties over a finite field can be jointly computed on encrypted data using an additively homomorphic encryption scheme. Our protocol is secure against malicious adversaries. To motivate and illustrate applications of this technique, we demonstrate an attack on a class of known protocols showing how to compromise location privacy of honest users by manipulating messages in protocols with additively homomorphic encryption. We demonstrate how to apply the technique in order to solve different problems in geometric applications. We evaluate our approach using a prototypical implementation. The results show that the added overhead of our approach is small compared to insecure outsourced multiplication.

Keywords: Location Privacy, Privacy-Enhancing Technologies, Secure Multi-Party Computation

1. Introduction

There has been an increase of the public awareness about the importance of privacy. This has become obvious with cases such as the disclosure by Edward Snowden [43] and the increased public interest in the Tor project [12]. Unfortunately, current best practice is not to address privacy concerns by design [8, 42,32,37]. It is by far more common that the end consumer has to send privacy-sensitive information to service providers in order to achieve a certain functionality, rather than that the service is using privacy-preserving technologies. A major research challenge is to enable privacy in services without hampering sought functionality and efficiency.

In recent years, much attention has been directed to secure computations distributed among several participants, a subfield of cryptography generally known as *Secure Multi-party Computation* (SMC). SMC has now been brought to the brink of being applicable to real world scenarios [4,3], although general purpose solutions with strong security guarantees are still too slow to be widely applied in practice.

This paper proposes a novel approach to jointly compute an arbitrary arithmetic formula using certain additively homomorphic encryption schemes, incurring very little overhead while maintaining privacy

0926-227X/17/$27.50 © 2017 – IOS Press and the authors. All rights reserved

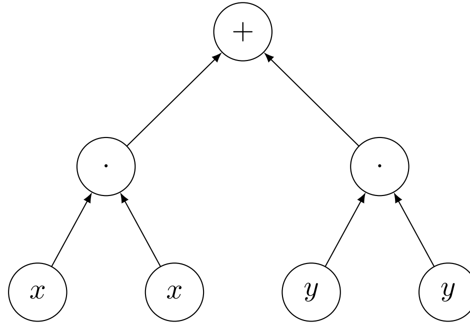

Fig. 1. Arithmetic formula computing $x^2 + y^2$.

against malicious adversaries. The solution is shown to be valuable as a vital complement to boost the security of a class of privacy-preserving protocols [13,40,22,39,45], where *Alice* queries *Bob* for a function over their combined inputs (see Figure 2). In such scenarios, it is common that *Bob* is intended to learn nothing at all, while still providing *Alice* with useful information such as whether a picture of a face matches a database [39,13] or whether two principals are close to each other [22,40,45]. The work presented in this paper allows such solutions to harden the attacker model from *honest-but-curious* to *malicious* attackers that do not necessarily follow the protocol (both attacker models are standard in SMC and are presented for instance in [18,33]).

Although some connections have been identified [35,40,22], the two communities of Privacy-preserving LBS and Secure Multi-Party Computations are still largely separated. One of the goals of this paper is to contribute to bridging the gap, in particular when it comes to rigorously improving the security of efficient protocols using additively homomorphic encryption in the presence of honest-but-curious adversaries, enabling them to also protect against malicious adversaries in an efficient manner.

Problem statement In general in secure two-party computation [33] one considers the case where two parties, *Alice* with inputs $\overrightarrow{x}$ and *Bob* with inputs $\overrightarrow{y}$, want to compute a functionality $f(\overrightarrow{x}, \overrightarrow{y}) = (g(\overrightarrow{x}, \overrightarrow{y}), h(\overrightarrow{x}, \overrightarrow{y}))$, where the procedure f yields a tuple in which *Alice*'s output is the first item and *Bob*'s output is the second item. For the scope of this work, h is always the empty string, and the inputs of both parties are in $\mathbb{F}_p$, such that $\forall x_i \in \overrightarrow{x} : x_i \in \mathbb{F}_p$ and $\forall y_i \in \overrightarrow{y} : y_i \in \mathbb{F}_p$. That is, *Alice* obtains the result of g whereas *Bob* observes nothing (as usual when using partial or full homomorphic encryption). For this reason, in the following we will refer only to $g(\overrightarrow{x}, \overrightarrow{y})$ as the functionality.

Moreover, we set $g(\overrightarrow{x}, \overrightarrow{y})$ to be an arbitrary arithmetic formula over $\overrightarrow{x}$ and $\overrightarrow{y}$ in the operations $(\cdot, +)$ of $\mathbb{F}_p$, that is an arithmetic circuit [41] that is also a graph with directed edges and no cycles, as the one depicted in Figure 1.

We assume as usual that both *Alice* and *Bob* want *privacy* of their inputs, as much as it is allowed by g. *Bob* is willing to reveal the final output of g, but not any intermediate results, or a different function g' that would compromise the privacy of his inputs. More precisely, we want a secure two-party computation in the malicious adversary model for a malicious *Alice* [33], as depicted in Figure 2. We assume that Bob has no interest in being malicious, although we do not provide mechanisms to enforce fairness or correcteness on the values computed by him.

Note that additions in the formula can be done correctly by *Bob* without the help of *Alice* when using an additively homomorphic encryption scheme. This holds also for all multiplications involving *Bob*'s input only, and multiplications of a ciphertext and a value known to *Bob*. The only operations outside of the scope of the additively homomorphic capabilities are multiplications involving inputs from *Alice*

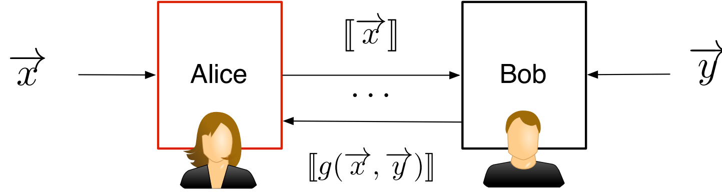

Fig. 2. High-level view of a 2-party computation based on homomorphic encryption, where $[\![\cdot]\!]$ denotes encryption under the public key of *Alice*.

only. For instance in Figure 1, *Bob* can not compute x^2 (assuming x is a private input from *Alice*). In this work therefore we focus on a protocol such that *Bob* can outsource such multiplications to *Alice* without disclosing the value of the operands, and such that if *Alice* does not cooperate, the final value of the arithmetic formula is corrupted and useless to her. This will allow us to show that our protocol is fully privacy-preserving in the malicious adversary model of SMC.

Fairness of the computation (that is, all parties receive their intended output) is out of scope for two reasons: it is impossible to guarantee this property for two-party computations in the malicious model [33]; further, *Bob* receives no output from the protocol by construction, which means that an early abortion of the protocol by *Alice* will only hamper fairness for herself.

Contributions The paper presents a general approach *BetterTrees* which lets *Alice* and *Bob* compute an arbitrary arithmetic function on their input while maintaining privacy even in the presence of malicious adversaries. The core of the solution is a multiplication protocol *BetterTimesMul*, using which *Bob* (who does not have the private key) can outsource multiplications using an additively homomorphic encryption scheme while asserting privacy of his inputs. BetterTimesMul provides *Bob* not only with the encrypted product but also the encryption of an assurance value (a field element $a \in \mathbb{F}_p$) which is a random value in $\mathbb{F}_p^*$ if *Alice* does not follow the protocol and an encryption of 0 otherwise. The assurance is added to the final output of g thus making the result useless to *Alice* if she tries to cheat. Our contribution thus brings the state-of-the art forward by efficiently giving *Bob* guarantees in the case that *Alice* is malicious.

We illustrate the usefulness of our approach for a class of protocols from the literature [13,40,22,39, 45]. All these protocols compute whether the distance between two vectors in the plane is less than a threshold and are secure against semi-honest adversaries. However, in the presence of malicious adversaries leakage of private information is possible. A solution using our technique is presented for these protocols, which plugs such leakage. We further demonstrate how to use BetterTrees to enforce a complex policy for location privacy, both to create a running implementation and to provide rigorous proofs of the resulting protocol. Moreover, we make our implementation fully available to the community [20].

Comparison to our previous work This paper revises and extends previous work that introduces privacy-assured outsourced multiplication [21,23] in various aspects. First, we seamlessly integrate multiple outputs [23] into the protocol, which is particularly useful for sequentially composing protocols, as we will demonstrate in the application scenario of private speed-constrained location proximity. We further develop the high-level syntax [23] that allows structured protocol design from the core primitives and show how it can be compiled into lower level verified constructions. Furthermore, we overhaul the formalization and proofs [21], as to accommodate compositional reasoning for protocols that leverage privacy-assured arithmetic formulas. Finally, we discuss the implication of using the system not only in fields but also for cryptographic schemes over rings, such as the Paillier cryptosystem, which provides

slightly different security guarantees. Finally, we provide benchmarks for the Paillier implementation of the framework.

Outline Section 2 introduces necessary notation and describes the BetterTimesMul protocol and its application to computing arbitrary arithmetic formulas. Section 3 presents the security guarantees in the malicious adversary setting. Section 5 presents benchmarks that allow one to estimate which impact the approach would have in comparison to only protecting against semi-honest adversaries. Section 6 positions this work in perspective to already published work. Finally, Section 7 summarizes the material presented in this paper. Before delving into details, a concrete application of the proposed solution is outlined in Section 4.

2. Secure Arithmetics for SMC

There are a variety of primitives for implementing SMC, including garbled circuits [44], partial [36] and fully homomorphic encryption schemes [15] among others. This section details BetterTrees, a construction without third parties, utilizing additively homomorphic encryption, which gives privacy guarantees against a malicious *Alice*. Further, being based on additively homomorphic cryptography, it supports storing intermediate values from previous computations, a central feature in many implementations.

This section proceeds by giving a background on additively homomorphic encryption, followed by describing the BetterTrees construction. First we describe BetterTrees using *BetterTimes-instructions*, which precisely define the workings of the solution. Following this, we also outline the python-compatible and more abstract *BetterTimes-syntax*, and show that from any program constructed using BetterTimes-syntax, a corresponding compilation into BetterTimes-instructions can be obtained. Thus, any program implemented using the BetterTimes-syntax can leverage the proofs we outline in the smaller language of BetterTimes-instructions.

2.1. Background

The solution proposed in this paper makes use of any additively homomorphic encryption scheme which provides semantic security and where the plaintext space is a field (such as the DGK Scheme [10]). For a definition of semantic security see [2].

Additively Homomorphic Encryption Schemes Here and henceforth, k is the private key belonging to *Alice* and K is the corresponding public key. Let the plaintext space $\mathcal{M}$ be isomorphic to the field $(\mathbb{Z}_p, \cdot, +)$ for some prime number p and the ciphertext space $\mathcal{C}$ such that encryption using public key K is a function $E : \mathcal{M} \to \mathcal{C}$ and decryption using a private key k is another function $D : \mathcal{C} \to \mathcal{M}$.

The vital homomorphic features which is used later in the paper is an addition function $\oplus : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$, a unary negation function $\neg : \mathcal{C} \to \mathcal{C}$, and a multiplication function $\odot : \mathcal{C} \times \mathcal{M} \to \mathcal{C}$, as shown in Equations 1–3.

$$E(m_1) \oplus E(m_2) = E(m_1 + m_2) \tag{1}$$

$$\neg E(m_1) = E(-m_1) \tag{2}$$

$$E(m_1) \odot m_2 = E(m_1 \cdot m_2) \tag{3}$$

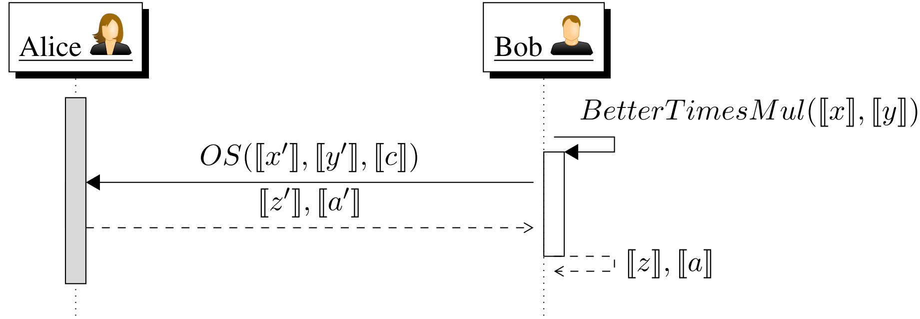

Fig. 3. Visualization of the assurance multiplication protocol

Note that in a finite field any non-zero element multiplied with a non-zero random element yields a non-zero uniformly distributed element. This is formalized in Equation (4), where $m_1 \in \mathcal{M}$, $m_2 \in \mathcal{M}$ and $\mathcal{M}^{\mathcal{U}}$ is a uniformly random distribution of all elements in $\mathcal{M} \setminus \{0\}$.

$$E(m_1) \odot \rho = \begin{cases} E(0) & \text{if } m_1 = 0 \\ E(\perp) & \text{otherwise} \end{cases}, \text{ with } \rho \in \mathcal{M}^{\mathcal{U}} \tag{4}$$

Syntax and conventions For readability, the operations $\oplus$, $\odot$, $\neg$, E and D are not annotated with the key associated to them, we assume they all use the usual k, K pair where *Alice* holds k, where only decryption has access to k and all instructions have access to K. The $\ominus$ symbol is used in the following to represent addition by a negated term. That is, $c_1 \oplus \neg c_2$ is written as $c_1 \ominus c_2$. For further brevity, a ciphertext c encrypting a plaintext p under the public key of *Alice* is denoted as $[\![p]\!]$.

The protocol description in Figure 4 and Figure 7 is given in the language pWHILE [1]. For the convenience of the reader a few constructs used in the paper are outlined here, but for details the reader is directed to [1]. $a \leftarrow b$ means assigning a value b to a variable a, while $a \xleftarrow{\$} [0..n]$ means assigning a random value between 0 and n to a.

2.2. BetterTrees: Arithmetic formulas through assured Multiplication

As previously discussed, our goal is a system which can compute any arithmetic formula in the presence of a malicious *Alice* (who holds the private key), without leaking any information derived from *Bob*'s inputs except the result of g. We call the solution BetterTrees. To show how to BetterTrees functions, we first outline the primary building block, BetterTimesMul.

2.2.1. Privacy-assured Outsourced Multiplication

The core of the solution is a novel outsourced multiplication protocol with privacy guarantees, BetterTimesMul. The protocol is visualized in Figure 3 and detailed in Figure 4. BetterTimesMul allows *Bob* to calculate a multiplication by outsourcing to *Alice*, while retaining an assurance value with which it is possible to make sure that *Alice* can learn no unintended information.

The principals interact once during BetterTimesMul, where *Bob* contacts *Alice* through the procedure OS (for outsource), defined in Figure 4. As a result of this interaction, *Bob* can compute a value $[\![z]\!]$ which corresponds to the encryption of the multiplication $x \cdot y$ if *Alice* computes the product honestly and an assurance value a which will be uniformly random if *Alice* does not comply with the protocol.

Proc. $BetterTimesMul([\![x]\!], [\![y]\!])$:
$c_a \xleftarrow{\$} \{0..p\}; c_m \xleftarrow{\$} \{0..p\};$
$b_x \xleftarrow{\$} \{0..p\}; b_y \xleftarrow{\$} \{0..p\};$
$\rho \xleftarrow{\$} \{1..p\};$
// Blind operands
$[\![x']\!] \leftarrow [\![x]\!] \oplus [\![b_x]\!]; [\![y']\!] \leftarrow [\![y]\!] \oplus [\![b_y]\!];$
// Create challenge
$[\![c]\!] \leftarrow ([\![x']\!] \odot c_m) \oplus [\![c_a]\!];$
// Outsource multiplication
$([\![z']\!], [\![a']\!]) \leftarrow OS([\![x']\!], [\![y']\!], [\![c]\!]);$
// Compute assurance value
$[\![a]\!] \leftarrow ([\![a']\!] \ominus [\![z']\!] \odot c_m \ominus [\![y']\!] \odot c_a) \odot \rho;$
// Un-blind multiplication
$[\![z]\!] \leftarrow [\![z']\!] \ominus ([\![x]\!] \odot b_y \oplus [\![y]\!] \odot b_x \oplus [\![b_x \cdot b_y]\!]);$
return $([\![a]\!], [\![z]\!]);$

Proc. $OS([\![x]\!], [\![y]\!], [\![c]\!])$:
return $((E(D([\![x]\!]) \cdot D([\![y]\!]), E(D([\![c]\!]) \cdot D([\![y]\!])));$

Fig. 4. The assured multiplication protocol

BetterTimesMul contains several random variables, here follows a brief explanation of their names to make the procedures easier to follow. The first two, c_a and c_m, serve to construct the challenge c used in the assurance. c_a and c_m are an additive and multiplicative component, respectively. The second pair, b_x and b_y, is used to blind the operands x and y, respectively, when outsourcing the multiplication. Finally, ρ is used to make sure that an assurance which does not match the supplied product causes a random offset of the final result.

Note that the assurance is only needed when outsourcing a multiplication. The blinding used in BetterTimesMul has also been presented and used by, among others, Kolesnikov et al. [27]. The construction using the challenges c_a and c_m yields the following computations in the plaintext, starting with the assurance value a in Equation (6). Through the procedure OS, *Alice* replies (in the plaintexts) as in Equation (5). Thus, assuming *Alice* is honest, we see that Equation (7) must hold.

$$a' = (x' \cdot c_m + c_a) \cdot y' = (x' \cdot y' \cdot c_m + y' \cdot c_a) \tag{5}$$

$$a = (a' - z' \cdot c_m - y' \cdot c_a) \cdot \rho = (x' \cdot y' \cdot c_m + y' \cdot c_a - z' \cdot c_m - y' \cdot c_a) \cdot \rho \tag{6}$$

$$a = (x' \cdot y' - z') \cdot c_m \cdot \rho \tag{7}$$

Since by assumption *Alice* is honest, $z' = x' \cdot y' \implies a = 0$. To see that this is the case if and only if *Alice* honest, see Section 3.

2.2.2. *Privacy-assured Arithmetic Formulas*

The following discusses how to construct arbitrary arithmetic formulas in the BetterTrees system using BetterTimesMul as described above. The general idea is to accumulate any errors caused by misbehavior by *Alice* using assurances a_j, one for each outsourced multiplication. The other operations require no assurances as they can be calculated locally by *Bob*. If *Alice* is dishonest during an outsourced multiplication, the corresponding assurance a_j is a uniformly random variable. Once an arithmetic formula has

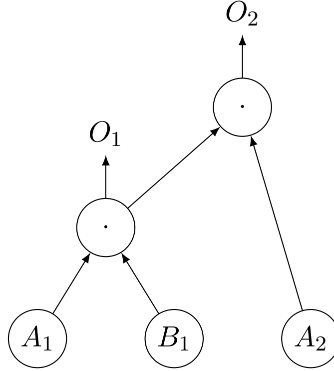

Fig. 5. Example of a formula with multiple outputs

been fully evaluated, and the result obtained as $[\![result]\!]$, *Bob* instead returns the value $[\![result]\!] \oplus \sum a_i$. The returned value is $[\![result]\!]$ if and only if *Alice* is honest, and the encryption of a uniformly random field element if she is dishonest.

To describe the precise workings of BetterTrees, the system is formally described using BetterTimes-instructions. BetterTimes-instructions are useful to create precise arguments about the security of the system, but cumbersome to use when constructing a program. To ease this problem, the more readable BetterTimes-syntax is presented later in this section as a subset of the Python programming language.

BetterTimes-instructions are constructed using a recursive data structure **Ins**, modeling an *instruction* representing an arbitrary arithmetic formula g. An instruction either contains an operation and two operands or a scalar. Formally, **Ins** $\in \{[o, l, r], x\}$, where o is the operator, l and r are the left- and right-hand side operands, and x is a scalar. The operands are nested instances of **Ins**. The operator is an enum-like variable, with four possible values $\{ADD$, SUB, MUL, $PMUL\}$. The scalar member holds a ciphertext or a plaintext. An instance ins of **Ins** is created using either $Ins(scalar)$, or $Ins(op, ins1, ins2)$. An instruction to compute the addition of two encrypted values $[\![x]\!]$ and $[\![y]\!]$ thus looks like as e.g.: $Ins(ADD, Ins([\![x]\!]), Ins([\![y]\!]))$. At the start of the protocol, *Bob* must collect *Alice*'s encrypted inputs, and hard-wire them into the algorithm. For an example, see Section 4.1.1.

BetterTrees allows for any intermediate values to be output from the computation and used outside of the circuit. To mark a value for output, a normal **Ins** instruction is annotated as $\mathbf{Ins}_O$. The assurance values of *all* outputs are combined, such that misbehavior at any time while the formula is computed yields $\perp$ for all outputs. Figure 5 shows a visualization of a formula with multiple outputs.

The core of the setup is the recursive procedure $binOp$, defined in Figure 7, which recursively computes an instruction including any nested instructions. The value returned by $binOp$ has the same structure as that of BetterTimesMul, but the assurance in the first part of the return value is now an accumulated value over all nested instructions. The main function, wrapping all functionality, is the $evaluate$ procedure, see Figure 7. Evaluate takes as parameter an algorithm modeled using nested instructions. Evaluate adds the assurance values and the result of the individual instructions, creating the final result – which is the output of g if and only if *Alice* is honest. For a visualization of messages exchanged and actions taken by each principal, see Figure 6.

The protocol resulting from $evaluate$ sets *Alice* as the initiating party, and she starts the protocol by sending her inputs to *Bob*. *Bob* then hardwires both his and *Alice*'s inputs into an instruction of nested operations, forming a graph with directed edges and no cycles like in Figure 8. Depending on g, *Bob* computes any local operations and executes BetterTimesMul as necessary, with as many iterations as nec-

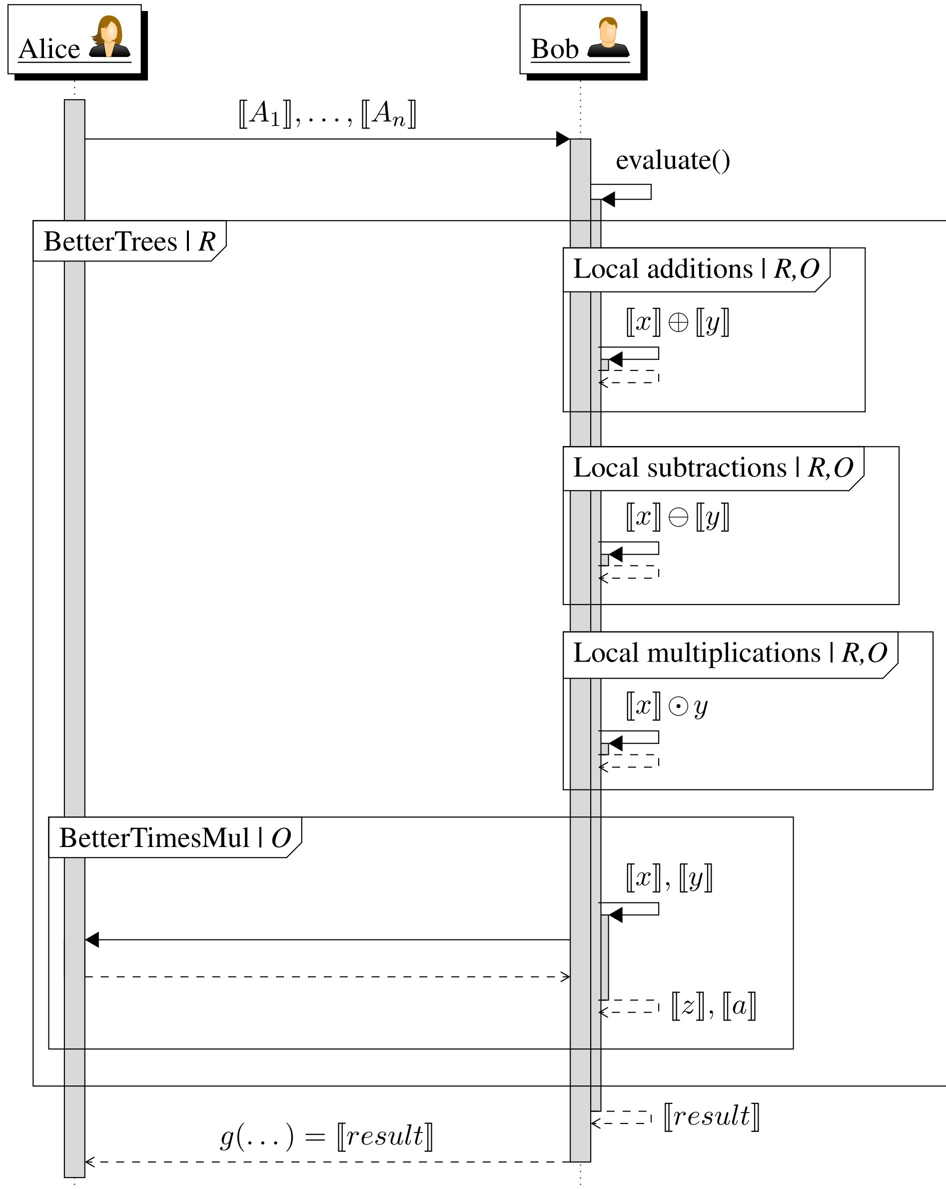

Fig. 6. Possible actions by each principal, where *R* and *O* means repeatable and optional, respectively.

essary. Finally, he computes the ciphertext $[\![result]\!]$. Since *Alice* by assumption is honest, $[\![result]\!]$ will hold the output of g (and would hold the encryption of a random element in $\mathbb{F}_p$ if *Alice* was dishonest).

The assurance values $[\![a_i]\!]$ are summed as the evaluation proceeds, to accumulate a final assurance $[\![a]\!]$. Finally, the assurance value is randomized and added to the result as $([\![result]\!] \oplus [\![a]\!]) \odot \rho$, with ρ random. When $evaluate$ encounters an Ins_O instruction, it computes the result and the assurance value as usual, but also saves the intermediate result $[\![z_i]\!]$ to a store S.

```
Proc. binOp(ins) :
if isScalar(ins) then :
    (a, z, outputs) ← (0, ins[0], []);
else :
    (a1, o1, out1) ← binOp(ins[1]);
    (a2, o2, out2) ← binOp(ins[2]);
    outputs ← out1 + out2;
    switch(ins[0]) :
        case ADD :
            (a, z) ← (a1 + a2, o1 + o2);
        case SUB :
            (a, z) ← (a1 + a2, o1 − o2);
        case PMUL :
            (a, z) ← (a1 + a2, o1 ∗ o2);
        case MUL :
            (a3, z) ← BetterTimesMul(o1, o2);
            a ← a1 + a2 + a3;
if isOutput(ins) then :
    return (a, z, outputs :: z);
else :
    return (a, z, outputs);
```

```
Proc. evaluate(alg) :
result ← [];
(a, z, outputs) ← binOp(alg);
for output in outputs do :
    ρ_i ←$ {1..p};
    result ← result :: output + (a ∗ ρ_i);
return result;
```

Fig. 7. The procedures to evaluate recursive instructions.

Finally, the array of the final result and all intermediate values is given as:

$$[(\llbracket result \rrbracket \oplus (\llbracket A \rrbracket \odot \rho)] :: [(fst(s) \oplus \llbracket A \rrbracket) \odot \rho_i \text{ for } s \in S]$$

Where ρ and all ρ_i are independent and uniformly random variables, and :: denotes array concatenation.

2.2.3. BetterTimes-syntax

This section details BetterTimes-syntax as a subset of python, which directly maps into BetterTimes-instructions. BetterTimes-syntax allows arithmetic formulas to be expressed in a readable and concise manner. The full source code is available online [20]. In short, BetterTimes-instructions are more easily represented using BetterTimes-syntax by normal addition, subtraction and multiplication operations. The operations are overloaded, and when used a datatructure is constructed in the background, which later can be translated into **Ins** objects and evaluated. The construction will be explained by working through an example, shown in Listing 1.

In Listing 1, we first create an instance of the used encryption scheme, DGK, and generate a keypair. Normally, the keypair would be generated by *Alice*, while the formula is evaluated by *Bob*. Note that the private key is not needed until we want to print the final output. Following, we define the encrypted inputs from *Alice* (you can assume these were previously sent over the network), and *Bob*'s inputs in plain. Either party may in theory have any number of plaintext and ciphertext inputs. *Alice* can send public parameters, and *Bob* may input ciphertexts obtained previously from *Alice* or third parties (encrypted

with *Alice*'s keys). After these initialization steps, we create a formula composer. The composer takes an encryption scheme, a key pair, and the inputs by the two parties.

The BetterTimes-syntax makes use of the built-in Python construct `with`. Traditionally, a with statement is used to manage the opening and closing of files as `with open('file.txt') as my_file_handle`. Here, we use it to initiate and finish the construction of a secure arithmetic formula, and the class `FormulaComposer` is used much like the `open` built-in function. The operations done with the inputs, and what the resulting outputs are, is defined by operating inside a `with` block. Upon leaving the `with` block, variables are no longer mutable, and the formula is fixed. There are four inputs provided to the formula, which we now call i_1, i_2, i_3 and i_4. At this point, we do not need to care about which ones originate from which party, and which ones are encrypted. There are two outputs, $i_1 * i_3$ and $(i_1 * i_3) + i_2 - i_4$. These outputs are marked using the `output()` method, by simply passing the variables that we want to output. The built-in `sum` function is used in the formula to show how easily one can make use of pre-existing arithmetic methods.

Listing 1: Example composition

```
scheme = DGK()
key_pair = scheme.keygen(1024)
public_key, private_key = key_pair

alice_inputs = [scheme.encrypt(public_key, 4), scheme.encrypt(public_key, 3)]
bob_inputs = [2, 1]
formula = FormulaComposer(scheme, public_key, alice_inputs + bob_inputs)

with formula as sf:
    i1, i2, i3, i4 = sf.inputs
    c = i1 * i3
    sf.output(c)  # 4 * 2 = 8
    sf.output(sum([c, i2, -i4]))  # (4*2) + 3 + (-1) = 10

assembled = formula.assemble()
outputs = evaluate(assembled)
print [scheme.decrypt(key_pair, output) for output in outputs] # prints [8, 10]
```

Finally, once the formula is fixed, we now use the `formula.assemple()` method to translate the internal representation of the formula and outputs into BetterTimes-instructions. Regardless of how the output is constructed, even when control flow primitives such as if statements and loops are used, the resulting computations can be represented through BetterTimes-instructions. The BetterTrees library includes an evaluator for BetterTimes-instructions, which is called through the `evaluate()` method. Thus, using the above construction gives all security properties achieved with BetterTimes-instructions (modulo implementation mistakes).

Note that using the BetterTimes-syntax, it is not necessary for the programmer to distinguish between MUL and $PMUL$ operations. Instead, the type of both operands is inspected to determine whether or not an outsourced multiplication is due. Further, it's not necessary to encrypt a value before using it in a formula. Instead, such conversions can be done automatically. For further examples, including networked applications, we refer the reader to the source code [20].

3. Privacy Guarantees of BetterTrees

The goal of this section is to show that the result of $evaluate$ as defined above is secure in the malicious adversary model for *Alice* (as depicted in Figure 2), following standard SMC security definitions.

3.1. Security Concepts

In the following we briefly recall some fundamental concepts from SMC that will be useful for the security guarantees discussion of Sect. 3.

Definition 1 (Negligible functions)**.** *A function* $\epsilon : \mathbb{N} \to \mathbb{R}$ *is said to be negligible if*

$$\forall\, c \in \mathbb{N}.\, \exists\, n_c \in \mathbb{N}.\, \forall_{n \geq n_c}\, |\epsilon(n)| \leq n^{-c}$$

That is, ϵ *decreases faster than the inverse of any polynomial.*

Definition 2 (Indistinguishability)**.** *The two random variables* $X(n, a)$ *and* $Y(n, a)$ *(where* n *is a security parameter and* a *represents the inputs to the protocol) are called computationally indistinguishable and denoted* $X \stackrel{c}{\equiv} Y$ *if for a probabilistic polynomial time (PPT) adversary* $\mathcal{A}$ *the following function is negligible:*

$$\delta(n) = |\Pr[\mathcal{A}(X(n, a)) = 1] - \Pr[\mathcal{A}(Y(n, a)) = 1]|$$

Ideal and Real Execution We construct our proofs using the idea of *"security as emulation of a real execution in the ideal model"*, following the definitions of Pinkas and Lindell [33]. In the following, this proof framework is recalled. We use two models, the IDEAL and the REAL. In the imaginary IDEAL model the parties interact only with a trusted third party, which ensures that the executed protocol matches exactly an implementation of the functionality, where parties cannot deviate from the protocol. In the REAL model, instead, a concrete instance of the protocol is considered. The goal is then to show that an attacker on the REAL model has no advantage over an attacker on the IDEAL model.

In the following, let $\overrightarrow{x} \in I_A$ and $\overrightarrow{y} \in I_B$ be the private inputs for two parties, and let $g(\overrightarrow{x}, \overrightarrow{y}) \in O_A \times O_B$ be the output of a functionality g, where O_A and O_B are arbitrary but fixed output spaces for both parties (for instance strings of bits of length n).

Formally, in the execution in the IDEAL model thus, an adversary $\mathcal{A}_{\text{IDEAL}}$ is controlling a corrupted party (*Alice* for the context of this paper). $\mathcal{A}_{\text{IDEAL}}$ tells the corrupted party (*Alice*) to either send the actual input of *Alice* (which can be read by $\mathcal{A}_{\text{IDEAL}}$) or another value of the same length as $\overrightarrow{x}$ to the trusted party. The trusted party then computes the output to be received by both parties. For the scope of this paper, *Bob* has no output, and *Alice* receives $g(\overrightarrow{x}, \overrightarrow{y})$. *Alice* forwards the result to $\mathcal{A}_{\text{IDEAL}}$. The original definition also handles the case when $\mathcal{A}_{\text{IDEAL}}$ wishes to abort the protocol. In the context of this paper, since the SMC solution is based on homomorphic encryption, *Bob* receives no output from the ideal functionality. Therefore, it does not make sense for the adversary to abort the protocol. Also, this means that fairness guarantees for *Bob* are out of scope, so abortions of the protocol by the simulator do not need to be accounted for. After the *ideal execution* of a functionality on inputs $(\overrightarrow{x}, \overrightarrow{y})$ $\mathcal{A}$ outputs an arbitrary PPT function on the private input of *Alice* and the output of the functionality $(\overrightarrow{x}, g(\overrightarrow{x}, \overrightarrow{y}))$. Formally thus:

$$\mathcal{A}_{\text{IDEAL}} : I_A \times O_A \to O_{\mathcal{A}}$$

We say that the distribution of the IDEAL model is:

$$\{\text{IDEAL}_{g,\mathcal{S}}(\overrightarrow{x}, \overrightarrow{y})\} = \{g(\overrightarrow{x}, \overrightarrow{y}), \mathcal{A}_{\text{IDEAL}}(\overrightarrow{x}, g(\overrightarrow{x}, \overrightarrow{y}))\}$$

The *real execution* of a concrete protocol π is rather intuitive, where $\mathcal{A}_{\text{REAL}}$ takes the place of the corrupted party and acts on their behalf. In this case:

$$\mathcal{A}_{\text{REAL}} : I_A \times view_\pi^A \times O_A \rightarrow O_\mathcal{A}$$

where $view_\pi^A$ are the intermediate values seen by $\mathcal{A}_{\text{REAL}}$ during the execution of π. We say that the distribution of the REAL model is:

$$\{\text{REAL}_{\pi,\mathcal{A}}(\overrightarrow{x}, \overrightarrow{y})\} = \{\overrightarrow{o}, \mathcal{A}_{\text{REAL}}(\overrightarrow{x}, view_\pi^A, \overrightarrow{o}))\}$$

Where $\overrightarrow{o}$ are the outputs to the corrupted party during an execution of π.

As previously mentioned, the goal of a proof in this framework is to show that an attacker in the real model is as effective as one in the ideal model. To do this, one constructs a simulator $\mathcal{S}$ of a $\mathcal{A}_{\text{REAL}}$ but which only interacts in the IDEAL model. If, for an arbitrary attacker against the real mode one can construct such a simulator and show that the distributions of the real and ideal models are indistinguishable, the attacker in the real model has no advantage than the one in the ideal model. Formally, it is required that for any adversary $\mathcal{A}_{\text{REAL}}$ against a protocol, there exists a simulator:

$$\mathcal{S} : I_A \times O_A \rightarrow O_\mathcal{A}$$

such that the distribution of the outputs of $\mathcal{S}$ and $\mathcal{A}_{\text{REAL}}$ are computationally indistinguishable:

$$\mathcal{S}(\overrightarrow{x}, g(\overrightarrow{x}, \overrightarrow{y})) \stackrel{c}{\equiv} \mathcal{A}_{\text{REAL}}(\overrightarrow{x}, view_\pi^A, \overrightarrow{o})\}$$

That is, a protocol π is privacy-preserving if it is possible to construct a concrete PPT $\mathcal{S}$ such that for every attacker $\mathcal{A}$ against the real protocol, its output is indistinguishable from the one of $\mathcal{A}$, using only the information available to the attacker (their inputs and the functionality output). If this is possible, an adversary does not learn anything apart from what is disclosed by the functionality by attacking π. This leads to the central privacy notion of this work, which is according to Definition 3, following the standard SMC security definitions of [33] against malicious adversaries. Since $\mathcal{A}_{\text{IDEAL}}$ henceforth is replaced by $\mathcal{S}$, $\mathcal{A}$ always refers to $\mathcal{A}_{\text{REAL}}$ in the following.

Definition 3 (Privacy definition). *A protocol π is said to privately implement a functionality g against malicious adversaries if for every adversary $\mathcal{A}$ against the protocol π, there exist a simulator $\mathcal{S}$ such that:*

$$\{\text{IDEAL}_{g,\mathcal{S}}(\overrightarrow{x}, \overrightarrow{y})\} \stackrel{c}{\equiv} \{\text{REAL}_{\pi,\mathcal{A}}(\overrightarrow{x}, \overrightarrow{y})\}$$

3.2. Proofs

Recall that a malicious *Alice* in possession of the private key can attack the privacy of the inputs of *Bob* by deviating from the original protocol (as discussed in Section 4 for a proximity calculation protocol). Intuitively, a malicious *Alice* will deviate from the protocol every time it fails to answer to the outsourced multiplication with the expected values z' and a' as defined in Figure 4. A deviation would be for example failing to multiply x' with y', in order to change the intended jointly computed arithmetic formula.

Formally, we set out to prove the following theorem, which is an instance of the general definition of [33] where the concrete SMC protocol π will depend on the arithmetic formula g to be jointly computed. In the following indistinguishability will be established with respect to the size p of the field $\mathbb{F}_p$ (p is thus the security parameter).

Theorem 1. *For a fixed but arbitrary arithmetic formula $g(\overrightarrow{x}, \overrightarrow{y})$ represented by a recursive instruction $\iota \in$ **Ins**, the protocol π resulting from $evaluate(\iota)$ is private according to Definition 3.*

In order to prove that the system is privacy-preserving, we need to construct a simulator which acts as the real attacker, but in the ideal model. We will do so by showing that any input not available to the simulator can be simulated using a computationally indistinguishable value. Recall that the attacker has to their disposal $\overrightarrow{x}$, $view_\pi^A$ and $\overrightarrow{o}$. Trivially, the input $\overrightarrow{x}$ can be simulated by using the input itself. The tricky parts are the view and the outputs. The view is explicit through the construction of $evaluate()$, but for clarity we also detail it here:

$$view_\pi^A = \{(x'_i, y'_i, c_i) | i \in \{0, \ldots, m\}\}$$

where m is the number of outsourced multiplications in π. That is, except for the inputs and the outputs, the only message are a triple for every outsourced multiplication, containing the blinded x and y values as well as the challenge c.

To show how to simulate the view and the outputs, we will use the fact that *Bob* (who is honest) controls a large portion of the computations, and that honest behavior is easy to simulate. The main points are captured in the following three lemmas. First, we show that the blinding used for each outsourced multiplication can be simulated using $\perp$ in Lemma 1. In Lemma 2, we look at the case when *Alice* tries to cheat during a multiplication to create an incorrect result where $z \neq x \cdot y$, and show that the assurance value then is $\perp$. Finally, we show in Lemma 3 we build on Lemma 2 to show that when *Alice* cheats, such that the out is not that of the functionality g, we can again use $\perp$ for simulating the output.

Lemma 1. *For a fixed but arbitrary arithmetic formula $g(\overrightarrow{x}, \overrightarrow{y})$ represented by a recursive instruction $\iota \in$ **Ins** against the protocol π resulting from $evaluate(\iota)$, all intermediate messages to* Alice *are independent and uniformly random.*

Proof. It follows from the procedures defined in Figure 4, that the intermediate values observed by an adversary $\mathcal{A}$ during the protocol execution are $[\![x']\!]$, $[\![y']\!]$ and $[\![c]\!]$, which are independently blinded and thus they are encryptions of uniformly random independent and indistinguishable from $([\![\perp]\!], [\![\perp]\!], [\![\perp]\!])$. □

Lemma 2. *In the outsourced multiplication protocol BetterTimesMul the assurance value* $\mathbf{a}$ *is equal to 0 if the protocol is followed, and is indistinguishable from $\perp$ otherwise.*

Proof. First recall the calculations from Figure 4:

$$[\![a]\!] \leftarrow \big([\![a']\!] \ominus [\![z']\!] \odot c_m \ominus [\![y']\!] \odot c_a\big) \odot \rho \qquad [\![z]\!] \leftarrow [\![z']\!] \ominus ([\![x]\!] \odot b_y \oplus [\![y]\!] \odot b_x \oplus [\![b_x \cdot b_y]\!])$$

Which in the plaintexts corresponds to:

$$a = \big(a' - z' \cdot c_m - y' \cdot c_a\big) \cdot \rho \qquad z = z' - (x \cdot b_y + y \cdot b_x + b_x \cdot b_y)$$

where a' and z' are produced by *Alice*. It is easy to see that if a' and z' are computed following the protocol, then $a = 0$ by construction.

To see that if *Alice* does not comply with the protocol then a is a randomly distributed non-zero element with very high probability, first note that there are three cases for non-compliance, either $z' \neq x' \cdot y'$, $a' \neq y' \cdot c$ or both. In any case of non-compliance, the goal of *Alice* is to construct a' and z' such that $a' - (z' \cdot c_m + y' \cdot c_a) = 0$ since otherwise by construction a will be random. From this we can easily see that then it must hold that $a' = (z' \cdot c_m + y' \cdot c_a)$.

Note that given $c = (x' \cdot c_m + c_a)$ (which is known by *Alice*), the probability of guessing c_m is at most $\epsilon = \frac{1}{2^p}$ where p is the size of the field, since multiplication is a random permutation and c_a is unknown and uniformly distributed.

Now by contradiction, let's assume that the probability of *Alice* of computing $a' = (z' \cdot c_m + y' \cdot c_a)$ with $z' \neq x' \cdot y'$ is bigger than ϵ. If this holds, then she can also compute:

$$\alpha = a' - c \cdot y' = (z' \cdot c_m + y' \cdot c_a) - (x' \cdot c_m + c_a) \cdot y'$$
$$\alpha = (z' - x' \cdot y') \cdot c_m$$

But then she could also compute $c_m = \alpha(z' - x' \cdot y')^{-1}$ with probability bigger than ϵ, since by hypothesis $z' \neq x' \cdot y'$ and thus $(z' - x' \cdot y') \in \mathbb{F}_p^*$ is invertible, which contradicts the fact that the probability of guessing c_m is smaller than ϵ. □

Lemma 3. *For a fixed but arbitrary arithmetic formula $g(\overrightarrow{x}, \overrightarrow{y})$ represented by a recursive instruction ι constructed using BetterTimes-syntax against the protocol π resulting from $evaluate(\iota)$, all output values of the protocol are an encryption of $\perp$ for a dishonest* Alice.

Proof. From Lemma 2, the decryption of an assurance value $[\![a]\!]$ is indistinguishable from $[\![\perp]\!]$ if an adversary is dishonest. Further, since by the construction of $evaluate()$, if any assurance value $a_i = \perp$, then the final assurance value is $\perp$. The final assurance value is added to the each output r_i as $r_i + a * \rho_i$ with an independently uniformly random ρ_i. Thus all outputs are indistinguishable from $[\![\perp]\!]$ if any a_i is $\perp$. □

Now, for the proof of Theorem 1:

Proof of Theorem 1. Without loss of generality, we assume that $\iota \in$ **Ins** has m instructions of type MUL and o outputs. We will distinguish two cases.

$\mathcal{A}$ follows the protocol As per Lemma 1, all m intermediate messages sent from *Bob* appear uniformly random to *Alice* (and independent) due to the fact that they are all of the type $r_i = (x', y', c)$ where each value is blinded. In the case when $\mathcal{A}$ complies with the protocol, the last message contains the correct output g, since *Bob* is an honest party. This implies that the output of $\mathcal{A}$ depends exclusively on $r_0, \ldots, r_m$ and $g(\overrightarrow{x}, \overrightarrow{y})$, so we can simulate an adversary as:

$$\mathcal{S} := \mathcal{A}(r_0^{\mathcal{S}}, \ldots, r_m^{\mathcal{S}}, g(\overrightarrow{x}, \overrightarrow{y}))$$

with $r_i^{\mathcal{S}} = (\perp, \perp, \perp)$ for $i \in \{0, \ldots, m\}$.

$\mathcal{A}$ does not follow the protocol Note that independently of the cheating strategy of $\mathcal{A}$, all m intermediate messages sent from *Bob* appear uniformly random to $\mathcal{A}$ since the blinding is done by *Bob* locally with randomization independent from $\mathcal{A}$'s inputs. Now, as a consequence of Lemma 3, if $\mathcal{A}$ does not follow the protocol for at least one of the outsourced multiplications, all outputs will be blinded by an independent (for each output) re-randomization of the accumulated assurance value, that is, all outputs are $\perp$. Therefore, the outputs, denoted $r_{m+1}, \ldots, r_{m+o}$, will contain an encryption of an independent random value. Therefore we can simulate this in the ideal model as:

$$\mathcal{S} := \mathcal{A}(r_0^{\mathcal{S}}, \ldots, r_m^{\mathcal{S}}, r_{m+1}^{\mathcal{S}}, \ldots, r_{m+o}^{\mathcal{S}})$$

with $r_i^{\mathcal{S}} = (\perp, \perp, \perp)$ for $i \in \{0, \ldots, m\}$ and $r_i^{\mathcal{S}} = \perp$ for $i \in \{m+1, \ldots, m+o\}$. With $\mathcal{S}$ defined as above, it is easy to see that $\mathcal{S}(\overrightarrow{x}, g(\overrightarrow{x}, \overrightarrow{y})) \stackrel{c}{\equiv} \mathcal{A}(\overrightarrow{x}, view_{\pi}^{A}, \overrightarrow{o})\}$.

Since the above two cases partition the set of all possible attackers, we conclude that we can simulate an arbitrary attacker.

□

3.3. *BetterTrees in Rings*

Given the advantages of partially homomorphic cryptographic constructions (such as the Paillier cryptosystem) where the plaintext space is a ring isomorphic to $\mathbb{Z}_n, n = p \cdot q$ (and not a field), a natural question arises: Is it possible to extend BetterTrees to work under rings $\mathbb{Z}_n$? Unfortunately, the answer is negative, since an attacker can leverage on non-invertible plaintexts to deviate from the protocol without losing information on the final computation. However, attackers are also limited in the amount of information they can learn, and thus in this setting it is possible to obtain a partial result, which gives slightly weaker privacy guarantees. Since such guarantees are weaker, they are less useful for the application domain considered in this article (location privacy). We refer the interested reader to Appendix B for a discussion of theses results.

4. Applications for proximity protocols

This section shows the usefulness of BetterTrees in two ways. First, we show how easily it can be used to upgrade a group of protocols which are secure only against semi-honest adversaries to being able to cope with malicious attackers. Secondly, we show a concrete application of BetterTrees to a speed-constrained proximity protocol by Hallgren et al. [23], both with respect to the description of the protocol but also to exemplify how to reuse the lemmas and theorems provided together with BetterTrees.

4.1. Attacks on proximity protocols

We illustrate the usefulness of our approach by an attack on a class of protocols from the literature [13,40,22,39,45], which compute whether the distance between two vectors in the plane is less than a threshold in a privacy-preserving manner. Popular applications of this algorithm are geometric identification and location proximity. For concreteness, this section focuses on the distance computation used in the *InnerCircle* protocol by Hallgren et al. [22]. The same attack also applies to the other representatives of the same class of protocols [13,40,39,45], but in many cases a successful exploit does not have as visible effects.

Hallgren et al. present a protocol for privacy-preserving location proximity. It is based on the fact that *Bob* can compute the euclidean distances from a point represented as three ciphertexts $[\![2x]\!]$, $[\![2y]\!]$ and $[\![x^2 + y^2]\!]$ to any other point known by *Bob* using additively homomorphic encryption (here $[\![\cdot]\!]$ stands for encryption under the public key of *Alice*). A problem with the approach is that *Bob* has no knowledge of how the ciphertexts are actually related, he sees three ciphertexts $[\![\alpha]\!]$, $[\![\beta]\!]$ and $[\![\gamma]\!]$. In the case that $\gamma \neq (\alpha/2)^2 + (\beta/2)^2$, subsequent computations may leak unwanted information. The distance is expressed as the (squared) distance as shown in Equation (8), computed homomorphically as shown in Equation (9) where only some of *Bob*'s inputs are needed in plaintext.

$$D = x_A^2 + y_A^2 + x_B^2 + y_B^2 - (2x_A x_B + 2y_A y_B) \tag{8}$$

$$[\![D]\!] = [\![x_A^2 + y_A^2]\!] \oplus [\![x_B^2 + y_B^2]\!] \ominus ([\![2x_A]\!] \odot x_B \oplus [\![2y_A]\!] \odot y_B) \tag{9}$$

Here, $\oplus$, $\ominus$ and $\odot$ are the homomorphic operations which in the plaintext space map to $+$, $-$ and $\cdot$ respectively (see Section 2.1). Now, by replacing the information sent by *Alice* by α, β and γ and observing that *Alice* can choose α and β arbitrarily, the expression becomes as in Equation (10):

$$D = x_B^2 + y_B^2 + \gamma + \alpha x_B + \beta y_B \tag{10}$$

The effects of the attack are very illustrative in [40,22,45]. In these works, *Bob* wants to return a boolean $b = (r^2 > D)$ indicating whether two principals are within r from each other. Thus the result given to *Alice* is the evaluation of the function $r^2 > x_B^2 + y_B^2 + \alpha x_B + \beta y_B + \gamma$. This is equivalent to the result of $r^2 - \gamma > x_B^2 + y_B^2 + \alpha x_B + \beta y_B$. Given that *Alice* knows r, she can encode it into the manipulated variables thus forcing the evaluation of $\delta > x_B^2 + y_B^2 + \alpha x_B + \beta y_B + \eta$, with $\gamma = r^2 - \delta - \eta$. By changing α, β and η, *Alice* can move the center of the queried area, and by tweaking δ she can dictate the size of the area, causing unwanted and potentially very serious information leakage (for instance by querying in arbitrarily located and precise areas such as buildings).

4.1.1. Securing protocols for euclidean distances

Based on the novel asserted multiplication presented in Section 2.2, a new structure for the protocols of Hallgren et al. can be constructed. Similar amendments can easily be constructed in similar form for other afflicted solutions [13,40,39,45]. Using the system proposed in this paper, it is possible to send only the encryption of x_A and y_A in the initial message, and securing the necessary squaring by means of BetterTrees.

An arithmetic formula which computes the distance directly using x_A, y_A, x_B and y_B is already defined in Equation (8). Now remains only to model this such that it can be computed by the system

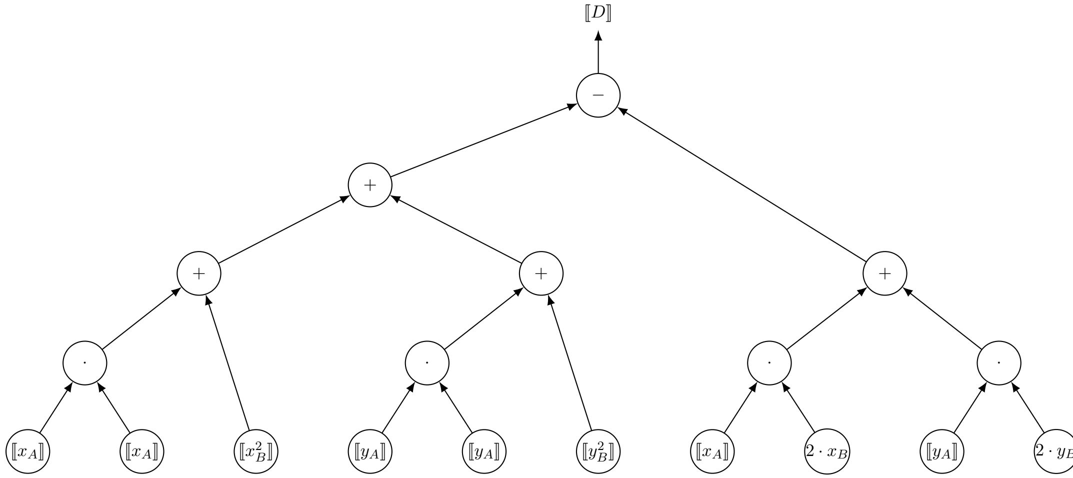

Fig. 8. Tree depicting computation of a secured version of the protocol.

presented later in this paper, after which the protocols can proceed to compute the proximity result as they would normally.

The result is an algorithm modeled using the recursive data structure Ins, which simply is passed to the procedure $evaluate$ by *Bob*. The formula of can be depicted as a tree as in Figure 8, for which the concrete instructions (instances of Ins) are spelled out in Equation (11). Per Theorem 1, this distance-computation is secure against malicious adversaries. Not however that additional effort may be needed to prove a full construction, if the following operations cannot be translated into instances of Ins.

$$
\begin{aligned}
&Ins(SUB,\\
&\qquad Ins(ADD,\\
&\qquad\quad Ins(ADD, Ins(MUL, Ins(\llbracket x_A \rrbracket), Ins(\llbracket x_A \rrbracket)), Ins(\llbracket x_B^2 \rrbracket)),\\
&\qquad\quad Ins(ADD, Ins(MUL, Ins(\llbracket y_A \rrbracket), Ins(\llbracket y_A \rrbracket)), Ins(\llbracket y_B^2 \rrbracket))\\
&\qquad),\\
&\qquad Ins(ADD,\\
&\qquad\quad Ins(PMUL, Ins(\llbracket x_A \rrbracket), Ins(2 \cdot x_B)),\\
&\qquad\quad Ins(PMUL, Ins(\llbracket y_A \rrbracket), Ins(2 \cdot y_B))\\
&\qquad),\\
&)
\end{aligned}
\tag{11}
$$

Of course, this formula can be more concisely represented using BetterTimes-syntax. One such representation is shown in Listing 2, which exactly corresponds to Equation (8). Note that an exponentiation a^b is denoted `a ** b` in Python.

Listing 2: Procedure for distance computation

```
def dist(x_a, y_a, x_b, y_b):
    return x_a ** 2 + y_a ** 2 + x_b ** 2 + y_b ** 2 - 2 * (x_a * x_b + y_a * y_b)
```

4.2. Simplification of MaxPace

In [23], Hallgren et al. show how to reduce the effectiveness of an attacker which try to infer additional data about the victim by issuing multiple location proximity queries. This section shows how to simplify their effort using the tools presented here (the original work was based on an older version of BetterTrees).

The core idea is a policy called MaxPace, which aims to reduce the velocity of the attacker, such that they may not query in rapid succession for proximity from positions that are far apart. The attacker can be enforced to behave as a user intuitively does – in patterns achieved by walking, riding a bike, using public transport, etc. The desired functionality for MaxPace is specified by Definition 4.

Definition 4 (Constrained speed querying functionality)**.** *The functionality of a speed-constraining functionality g is a function from queries to responses: $g : Q \rightarrow L$.*

$$g(q_1, \ldots, q_m)[i] = \begin{cases} \bot \quad \textit{if}\ \exists_{j<i} : \frac{dist(p_j, p_{j+1})}{time(q_j) - time(q_{j+1})} > h \\ inProx(p_i, p_b, r) \quad \textit{otherwise} \end{cases}$$

where $q_i = (p_i, t_i)$ and p_b is the position of Bob.

The MaxPace can be implemented in a straightforward way using a trusted third party who stores and manages location information for all users who are utilizing the service. Any already existing service can easily deploy MaxPace as an additional privacy measure. Many applications scenarios lack a natural third party that can be trusted, and a decentralized trust-model has obvious benefits as compared to giving location information to third parties. Services are usually not deployed in a decentralized manner without trusted parties, as for most application scenarios there are no ad-hoc solutions readily available. Hence, in these situations a tool like BetterTrees comes in handy.

Listing 3: Request handling using DecentMP

```
def mpRequest(ev,c_xA,c_yA,xB,yB,h,r,cache):
  formula = SecureFormula(ev, h,r, c_xA, c_yA, xB, yB,
              cache['a'], cache['t'], cache['x'], cache['y'])
  with formula as sf:
    c_xA,c_yA,xB,yB,h,r,ct,c_ca,c_cx,c_cy = sf.inputs
    t = now()
    pr = proximity(c_xA, c_yA, xB, yB, r)
    if 'x' in cache:
      v = speed(c_xA,c_yA,c_cx,c_cy,h,t-ct)
      alpha = random(1, k) * (v + c_ca)
      sf.output(pr + alpha)
      sf.output(alpha)
    else:
      sf.output(pr)

  out = formula.evaluate()
  c_result = out[0]
```

```
cache['a'] = out[1] if 'x' in cache else 0
cache['t'] = t
cache['x'] = c_xA
cache['y'] = c_yA
return c_result
```

The protocol devised to enforce MaxPace is called DecentMP, and is represented using BetterTimes-syntax in Listing 3. For further details on any functions called, see Appendix A. This means that the procedure is executed by *Bob*, while operations carried out by *Alice* are implicitly determined through the BetterTrees system.

For the first run, the protocol simply returns the proximity result and caches the query's position and time. *Bob* also initializes a special cache value a which is used to accumulate all speed threshold checks. For following requests, the speed threshold v is combined with the accumulated speed threshold. By adding the proximity result to the accumulated speed threshold, *Bob* constructs `c_result`. Note that all values depending on *Alice*'s inputs are encrypted and not readable by *Bob*.

Bearing the functionality shown in Definition 4 and the protocol resulting from Listing 3 in mind, we now present the main theorem of the original work [23], in order to show how to make use of the results presented in this work to create a simple proof of a construction such as DecentMP. The privacy-guarantees sought for DecentMP are captured by Theorem 2.

Theorem 2 (Privacy guarantees of DecentMP)**.** *The protocol π resulting from evaluating the program Listing 3 implements the functionality of Definition 4 privately according to Definition 3.*

The intuition behind the proof of Theorem 2 is as follows. DecentMP defines α as an arithmetic formula. From the security guarantees of BetterTrees, it follows that a malicious *Alice* that tampers with the protocol at any point will cause α to encrypt $\perp$. This in turn will cause the proximity result sent to *Alice* to be random, and cause `cache['a']` to be updated just as in the case where *Alice* does not respect the MaxPace speed policy, making subsequent location responses yield an encryption of $\perp$. Thus, if *Alice* would tamper with the protocol to try to learn more about the private inputs of *Bob* than allowed by the arithmetic formula (the functionality), BetterTrees guarantees that she instead receives a fresh uniformly random value.

Proof of Theorem 2. After performing m location queries *Alice* has observed the intermediate values in each query q_i and the respective location response l_i by *Bob*. From Lemma 3, and by construction of the `cache['a']`, it follows that if *Alice* cheats for the first time when jointly computing l_i with *Bob*, then $l_i = \perp$ and $\forall_{j>i}\ l_j = \perp$. Further, from Lemma 1, it follows directly that all intermediate values in a joint computation, denoted by $\overrightarrow{v}_i$, are equal to $\perp$. Without loss of generality, let's assume that a class of malicious adversaries $\mathcal{A}_x$ are dishonest when jointly computing the location response l_x. The output of any such malicious $\mathcal{A}_x$ against DecentMP can be simulated by a simulator $\mathcal{S}_x$ that outputs:

$$\mathcal{S}_x(q_1, \ldots, q_m, l_1, \cdots l_m) =$$
$$\mathcal{A}_x(q_1, \ldots, q_m, l'_1, \cdots l'_m, \overrightarrow{v}_1, \cdots \overrightarrow{v}_m)$$

Table 1

Benchmarks for outsourced multiplication

Plaintext space	1024 bits			2048 bits		
	This approach	Naive approach	Extra work	This approach	Naive approach	Extra work
DGK						
2^8	6.400	4.017	59.32%	30.052	19.484	54.24%
2^{24}	6.538	4.100	59.46%	30.578	19.801	54.43%
Paillier						
Equal to keysize	41.811	24.982	67.36%	292.333	175.325	66.74%

The outputs l'_i corresponding to the view of $\mathcal{A}$ in a real execution are easy to simulate, as they can be computed using only the inputs through:

$$l'_i = \begin{cases} \perp & \text{if } i \geq x \\ l_i & \text{otherwise} \end{cases}$$

And the intermediate messages can be simulated as $\overrightarrow{v}_j = \perp, \cdots, \perp$ as per Lemma 1.

□

5. Implementation and Benchmarks

5.1. Multiplication cost evaluation

Comparison against insecure outsourced multiplication The approach has been implemented in Python using the GMP [14] arithmetic library. The implementation has been benchmarked to show the impact of using our approach compared to the more common approach of naive outsourced multiplications. In the naive approach, *Alice* is honest-but-curious, and the operands are therefore only blinded. For this implementation, the DGK [10] cryptosystem was used.

Table 1 shows time in milliseconds for different sizes of plaintexts and keys for the two cases when outsourced multiplication is performed using BetterTimesMul, or naively. The difference between the two approaches is a small factor of about 1.5 for both key sizes, though slightly smaller for the larger keys. The factor is only marginally increasing as the plaintext space grows from 2^8 to 2^{24}.

The benchmarked time shows only the processing time for each multiplication. The communication overhead is exactly twice for our approach as compared to the naive solution.

Paillier We have also included in Table 1 results on the implementation of BetterTimes using Paillier with a key of size of 1024 bits and 2048 bits. Although as discussed before, the obtained security guarantees are different in this case, there are advantages in using Paillier when the plaintext space needs to be bigger than what usually supported by DGK.

5.2. Comparison with Garbled Circuits

It is difficult to directly compare our work with state-of-the-art implementations of Garbled Circuits for two main reasons: on the one hand, most implementations such as [25] provide only guarantees for semi-honest players; on the other hand, recent implementations of garbled circuits are the product of decades

of optimizations, and thus usally are written in C/C++, whereas our proof-of-concept implementation is written in Python.

However, it is possible to at least in principle compare advantages and disadvantages of both approaches in the following aspects:

Cost of a single multiplication We have implemented a circuit in FastGC [25] for 24-bit numbers, which can be performed in 332ms (in the same hardware as our benchmarks) which is two orders of magnitude slower than BetterTimesMul.

Bandwidth The bandwidth required for a GC is a factor of the number of gates of the circuit. In the case of a single multiplication this was 1084KB[1]. The bandwith cost of one multiplication of BetterTimesMul is 10KB for a 2048-bit key. Moreover note that a whole new GC must be generated and shared every time a new computation takes place between two-parties, whereas for BetterTrees only the inputs of Alice, the intermediate multiplication steps and the final result need to be exchanged over the network.

Integer comparisons It is well known that although partially homomorphic encryption can be advantegeous to perform arithmetic operations, it suffers from efficient constructions to perform certain common operations such as less-than comparisons. This is the main motivation for hybrid GC and homomorphic approaches such as TASTY [24]. Note that however, an inefficient (quadratic) algorithm to perform comparisons such as the one proposed in [22] can outperform a constant GC solution for small values (comparing a value a with r for $r < 100$ using Paillier takes less than 1.6ms, which is the constant value of a comparison for FastGC). Since the quadratic comparison in [22] is done without the need of outsourced multiplications, the time would be the same for a BetterTimesMul implementation. Also the comparison algorithm of [22] has been shown to be highly parallelizable, as we will discuss in the following for the InnerCircle case. In contrast, GC-based or hybrid GC and homomorphic encryption protocols are not known to be easily parallelizable.

In sum, a comparison with a modern GC implementations is heavily application dependent: for computations involving several multiplications and few or no integer comparisons, BetterTrees can be advantageous both in computation time and bandwidth; however for other computations GC might be the better choice.

5.3. Benchmarking MaxPace and InnerCircle

Note that the code snipets presented above for distance computation and the exemplary application MaxPace can directly be compiled into a secure 2-party protocol using BetterTimesMul. In the following we report a summary of benchmarking experiments for both protocols.

InnerCircle We have benchmarked an implementation of InnerCircle using DKG and a 1024 key, up to the distance calculation phase. An implementation with BetterTimesMul takes 96ms to run, whereas an implementation with insecure outsourcing would take 15ms. For the proximity response phase, the results depend strongly on the distance bound r and the number of cores used in the computation, since the proximity response phase is easy to parallelize. For instance, using Paillier and a 1024 bits key, the running time of the proximity phase for $r \leq 100$ remains below 1.6 seconds, which is faster than a FastGC implementation. In Table 2 we report experiments on the speed-up factor by a parallel implementation of the proximity response phase [22], which are close to the maximum theoretical speed-up. Moreover, the

[1]By using the tool in [25].

Fig. 9. Different speeds and proximity thresholds

bandwidth needed for $r = 100$ and a 2048-bit Paillier key is around 1.3MB, whereas for FastGC 17MB are needed.

Table 2
Results using different number of threads

Threads used	6	8	10
Speedup	51.29%	71.20%	76.76%
Max theoretical	66.67%	75.00%	80.00%

MaxPace Figure 9 visualizes how the time of a single protocol execution time is affected by different configurations of h and r. Benchmarks were carried out with a key of sizes 2048 bits, and plaintext space 22 bits. Table 3 shows on the other hand how different values of r and h affect communication. Communication cost ranges from 166 kilobytes to 12 megabytes. Note that although 12MB might seem big, as most modern devices and networks can handle high-quality video streaming, all results are within practical applicability. This also is still below the 17MB needed for a single proximity query using FastGC.

Table 3
Communication cost in kilobytes (messages)

Activity	Proximity Threshold (meters)				Messages
	10	15	50	100	
Walking	166	610	2050	7392	11
Running	196	640	2080	7422	17
Cycling	286	730	2170	7512	35
Bus	1076	1520	2960	8302	193
Car	4706	5150	6590	11932	919

6. Related Work

There are three current approaches to compute an arbitrary formula in the two-party setting in the presence of malicious adversaries, Fully Homomorphic Encryption, Enhanced Garbled Circuits and Zero-knowledge proofs.

FHE is by far the most inefficient approach, and its use is often considered not feasible due to the heavy resource consumption. We do not consider FHE a viable alternative to additively homomorphic encryption for practical applications. Garbled Circuits is an excellent tool for boolean circuits, but does not perform as well for arithmetic circuits as approaches built on homomorphic encryption. Zero-knowledge proofs could be used instead of the proposed approach, but at the cost of more computations and/or round trips.

6.1. Zero-knowledge Proofs

The technique which resembles BetterTimesMul the most is *Zero-Knowledge* (ZK) proofs. Any statement in NP can be proven using generic, though inefficient, ZK (Goldreich et al. [19]). However, to the best of our knowledge it is not straightforward to constructively devise such a scheme for a given additively homomorphic cryptosystem. Our solution in contrast does not require *Bob* to be able to verify whether a multiplication is correct, but by construction will render the final computation result useless to malicious adversaries.

In a nutshell, the novelty as compared to zero-knowledge proofs is based on the simple realization that *Bob* does not need to know whether *Alice* is cheating or not in order to assure the correctness of the final computation and the privacy of his inputs, which decreases the number of round-trips that such a verification step implies. This is a special case of the *conditional disclosure of secrets* introduced by Gertner et al. [17], where a secret is disclosed using SMC only if some condition is met. In our case, the condition is that $z_i = x_i \cdot y_i$ for each multiplication in the formula, and the secret is the output of g.

Some protocols in the literature can be used efficiently for proving correct multiplications, with only one additional round trip. One such is the Chaum-Pedersen protocol [7], which however is not trivially applicable to an arbitrary encryption scheme. Another interesting solution was introduced by Damgård and Jurik [11], but which is constructed specifically for the Damgård-Jurik cryptosystem.

Thought there are some homomorphic schemes able to handle both additions and multiplications, to the best of our knowledge, there is no previous solution to accomplish secure outsourced multiplications for additively homomorphic encryption in the malicious model without the use of zero-knowledge proofs (with the exception of second degree-functions, see [6]).

6.2. Secure Multi-party Computations

There are two main categories for private remote computations: Homomorphic Encryption and Garbled Circuits. Through recent research they are both near practical applicability (see [29,25,30] and [5,25, 16]). However, which of the two approaches to choose is typically application-dependent [31,28]. Our approach brings state-of-the-art SMC solutions based on additively homomorphic cryptographic systems forward by protecting against malicious adversaries when outsourcing multiplications, while remaining strongly competitive to the efficient though less secure approaches which currently are popular examples.

There are several works that combine the use of an additively homomorphic scheme with secret sharing, to compute multiplications securely using threshold encryption. This line of work stems from the SMC schemes developed by Cramer et al. [9]. Note that such approaches are secure only against malicious *minorities*, and are not directly applicable in scenarios with only two parties.

To compare against GC-solutions which can compute arbitrary formulas, some experiments using FastGC, a Garbled Circuit framework by Huang et al. [25] were conducted and are reported in detail in Section 5. Any arithmetic circuit can be expressed as a binary circuit, and vice versa[15]. In this framework for arbitrary computations, integer multiplication of 24-bit numbers needed 332 ms to finish, approximately 5078% slower than BetterTrees. Note however that FastGC is only secure in the honest-but-curious model, and thus not as secure as the approach presented in this paper. Further work exists in the direction of efficiently providing security against malicious adversaries by the authors of FastGC [26], however where one bit of the input is leaked. Moreover, work on optimizing garbled-circuits in the honest-but-curious model also exists, e.g. recently [34], but so far without enough speedup that it can compare to additively homomorphic encryption for privately computing arithmetic formulas.

7. Conclusions

We have presented a protocol for outsourcing multiplications and have shown how to use it construct a system for computation of arbitrary arithmetic formulas with strong privacy guarantees. We have shown that the construction is secure in the malicious adversary model and that the overhead of using the approach is a small constant factor.

The need for such a protocol is justified by the format attacks we have unveiled in known protocols, and presented a concrete exploit targeting [45] where we can alter the format of a message and gain more than the intended amount of location information. We have made a case for using a more realistic attacker model and identified examples from the literature which are vulnerable to this stronger attacker, while also showing how to amend such vulnerabilities.

A key feature of our approach is its compositionality. We we seamlessly integrate multiple outputs, allowing us to sequentially compose protocols. The new high-level syntax and proof framework prove indispensable when we show how our approach applies to the application domain of speed-constrained location proximity. Leveraging the compositionality, we construct the proximity protocol from core primitives while obtaining privacy guarantees by compositional reasoning.

We provide benchmarks that compare our exemplary applications (InnerCircle and MaxPace) to implementations based on Garbled Circuits. We discuss security guarantees for the particular case of an implementation over rings, and make our implementation fully available to the community. In sum our approach provides strong security guarantees when implemented over fields (for instance in the DKG cryptosystem) and has competitive efficiency and bandwidth performance against other 2-party protocols. However the general case depends strongly on the type of computation performed and the range of the variables involved. Moreover note that the security guarantees given can be summarized as strong privacy guarantees for both parties whenever any party is malicious, but we offer no correctness guarantees on the result of the computation when Bob is malicious.

As future work we plan to investigate the non-trivial task of applying closely related primitives (such as Zero-Knowledge constructions [7] and Threshold Encryption [9]) to achieve the same security guarantees, and benchmark those solutions to compare them to BetterTrees.

Acknowledgments Thanks are due to Allen Au for the useful comments. This work was funded by the European Community under the ProSecuToR project and the Swedish research agencies SSF and VR.

References

[1] G. Barthe, B. Grégoire, and S. Z. Béguelin. Formal certification of code-based cryptographic proofs. In Z. Shao and B. C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 90–101. ACM, 2009.

[2] M. Bellare, A. Boldyreva, and S. Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In B. Preneel, editor, *Advances in Cryptology - EUROCRYPT 2000, International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 14-18, 2000, Proceeding*, volume 1807 of *Lecture Notes in Computer Science*, pages 259–274. Springer, 2000.

[3] D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In S. Jajodia and J. López, editors, *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 192–206. Springer, 2008.

[4] P. Bogetoft, D. L. Christensen, I. Damgård, M. Geisler, T. P. Jakobsen, M. Krøigaard, J. D. Nielsen, J. B. Nielsen, K. Nielsen, J. Pagter, M. I. Schwartzbach, and T. Toft. Secure multiparty computation goes live. In R. Dingledine and P. Golle, editors, *Financial Cryptography and Data Security, 13th International Conference, FC 2009, Accra Beach,*

Barbados, February 23-26, 2009. Revised Selected Papers, volume 5628 of *Lecture Notes in Computer Science*, pages 325–343. Springer, 2009.

[5] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. Fully homomorphic encryption without bootstrapping. *Electronic Colloquium on Computational Complexity (ECCC)*, 18:111, 2011.

[6] D. Catalano and D. Fiore. Using linearly-homomorphic encryption to evaluate degree-2 functions on encrypted data. In Ray et al. [38], pages 1518–1529.

[7] D. Chaum and T. P. Pedersen. Wallet databases with observers. In E. F. Brickell, editor, *Advances in Cryptology - CRYPTO '92, 12th Annual International Cryptology Conference, Santa Barbara, California, USA, August 16-20, 1992, Proceedings*, volume 740 of *Lecture Notes in Computer Science*, pages 89–105. Springer, 1992.

[8] D. Coldewey. "Girls Around Me" Creeper App Just Might Get People To Pay Attention To Privacy Settings. `http://techcrunch.com/2012/03/30/girls-around-me-creeper-app-just-might-get-people-to-pay-attention-to-privacy-settings/`, Mar. 2012.

[9] R. Cramer, I. Damgård, and J. B. Nielsen. Multiparty computation from threshold homomorphic encryption. In B. Pfitzmann, editor, *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, volume 2045 of *Lecture Notes in Computer Science*, pages 280–299. Springer, 2001.

[10] I. Damgård, M. Geisler, and M. Krøigaard. Efficient and secure comparison for on-line auctions. In J. Pieprzyk, H. Ghodosi, and E. Dawson, editors, *Information Security and Privacy, 12th Australasian Conference, ACISP 2007, Townsville, Australia, July 2-4, 2007, Proceedings*, volume 4586 of *Lecture Notes in Computer Science*, pages 416–430. Springer, 2007.

[11] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In K. Kim, editor, *Public Key Cryptography, 4th International Workshop on Practice and Theory in Public Key Cryptography, PKC 2001, Cheju Island, Korea, February 13-15, 2001, Proceedings*, volume 1992 of *Lecture Notes in Computer Science*, pages 119–136. Springer, 2001.

[12] R. Dingledine, N. Mathewson, and P. F. Syverson. Tor: The second-generation onion router. In M. Blaze, editor, *Proceedings of the 13th USENIX Security Symposium, August 9-13, 2004, San Diego, CA, USA*, pages 303–320. USENIX, 2004.

[13] Z. Erkin, M. Franz, J. Guajardo, S. Katzenbeisser, I. Lagendijk, and T. Toft. Privacy-preserving face recognition. In I. Goldberg and M. J. Atallah, editors, *Privacy Enhancing Technologies, 9th International Symposium, PETS 2009, Seattle, WA, USA, August 5-7, 2009. Proceedings*, volume 5672 of *Lecture Notes in Computer Science*, pages 235–253. Springer, 2009.

[14] Free Software Foundation. The gnu multiple precision arithmetic library. `http://gmplib.org/`, 1991-2013.

[15] C. Gentry. Fully homomorphic encryption using ideal lattices. In M. Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178. ACM, 2009.

[16] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2013.

[17] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. *J. Comput. Syst. Sci.*, 60(3):592–629, 2000.

[18] O. Goldreich. *The Foundations of Cryptography - Volume 2, Basic Applications*. Cambridge University Press, 2004.

[19] O. Goldreich, S. Micali, and A. Wigderson. Proofs that yield nothing but their validity for all languages in NP have zero-knowledge proof systems. *J. ACM*, 38(3):691–729, 1991.

[20] P. Hallgren. Bettertimes. `https://bitbucket.org/hallgrep/bettertimes`, 2017.

[21] P. A. Hallgren, M. Ochoa, and A. Sabelfeld. Bettertimes - privacy-assured outsourced multiplications for additively homomorphic encryption on finite fields. In M. H. Au and A. Miyaji, editors, *Provable Security - 9th International Conference, ProvSec 2015, Kanazawa, Japan, November 24-26, 2015, Proceedings*, volume 9451 of *Lecture Notes in Computer Science*, pages 291–309. Springer, 2015.

[22] P. A. Hallgren, M. Ochoa, and A. Sabelfeld. Innercircle: A parallelizable decentralized privacy-preserving location proximity protocol. In A. A. Ghorbani, V. Torra, H. Hisil, A. Miri, A. Koltuksuz, J. Zhang, M. Sensoy, J. García-Alfaro, and I. Zincir, editors, *13th Annual Conference on Privacy, Security and Trust, PST 2015, Izmir, Turkey, July 21-23, 2015*, pages 1–6. IEEE Computer Society, 2015.

[23] P. A. Hallgren, M. Ochoa, and A. Sabelfeld. Maxpace: Speed-constrained location queries. In *2016 IEEE Conference on Communications and Network Security, CNS 2016, Philadelphia, PA, USA, October 17-19, 2016*, pages 136–144. IEEE, 2016.

[24] W. Henecka, A.-R. Sadeghi, T. Schneider, I. Wehrenberg, et al. Tasty: tool for automating secure two-party computations. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 451–462. ACM, 2010.

[25] Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011.

[26] Y. Huang, J. Katz, and D. Evans. Quid-pro-quo-tocols: Strengthening semi-honest protocols with dual execution. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 272–284. IEEE Computer Society, 2012.

[27] V. Kolesnikov, A. Sadeghi, and T. Schneider. From dust to dawn: Practically efficient two-party secure function evaluation protocols and their modular design. *IACR Cryptology ePrint Archive*, 2010:79, 2010.

[28] V. Kolesnikov, A. Sadeghi, and T. Schneider. A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design. *Journal of Computer Security*, 21(2):283–315, 2013.

[29] V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In L. Aceto, I. Damgård, L. A. Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[30] B. Kreuter, A. Shelat, and C. Shen. Billion-gate secure computation with malicious adversaries. In T. Kohno, editor, *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012*, pages 285–300. USENIX Association, 2012.

[31] R. L. Lagendijk, Z. Erkin, and M. Barni. Encrypted signal processing for privacy protection: Conveying the utility of homomorphic encryption and multiparty computation. *IEEE Signal Process. Mag.*, 30(1):82–105, 2013.

[32] M. Li, H. Zhu, Z. Gao, S. Chen, L. Yu, S. Hu, and K. Ren. All your location are belong to us: breaking mobile social networks for automated user location tracking. In J. Wu, X. Cheng, X. Li, and S. Sarkar, editors, *The Fifteenth ACM International Symposium on Mobile Ad Hoc Networking and Computing, MobiHoc'14, Philadelphia, PA, USA, August 11-14, 2014*, pages 43–52. ACM, 2014.

[33] Y. Lindell and B. Pinkas. Secure multiparty computation for privacy-preserving data mining. *IACR Cryptology ePrint Archive*, 2008:197, 2008.

[34] C. Liu, X. S. Wang, K. Nayak, Y. Huang, and E. Shi. Oblivm: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 359–376. IEEE Computer Society, 2015.

[35] A. Narayanan, N. Thiagarajan, M. Lakhani, M. Hamburg, and D. Boneh. Location privacy via private proximity testing. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[36] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In J. Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999.

[37] I. Polakis, G. Argyros, T. Petsios, S. Sivakorn, and A. D. Keromytis. Where's wally?: Precise user discovery attacks in location proximity services. In Ray et al. [38], pages 817–828.

[38] I. Ray, N. Li, and C. Kruegel, editors. *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*. ACM, 2015.

[39] A. Sadeghi, T. Schneider, and I. Wehrenberg. Efficient privacy-preserving face recognition. In D. H. Lee and S. Hong, editors, *Information, Security and Cryptology - ICISC 2009, 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers*, volume 5984 of *Lecture Notes in Computer Science*, pages 229–244. Springer, 2009.

[40] J. Sedenka and P. Gasti. Privacy-preserving distance computation and proximity testing on earth, done right. In S. Moriai, T. Jaeger, and K. Sakurai, editors, *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, pages 99–110. ACM, 2014.

[41] A. Shpilka and A. Yehudayoff. Arithmetic circuits: A survey of recent results and open questions. *Foundations and Trends in Theoretical Computer Science*, 5(3-4):207–388, 2010.

[42] M. Veytsman. How i was able to track the location of any tinder user. `http://blog.includesecurity.com/2014/02/how-i-was-able-to-track-location-of-any.html`, Feb. 2014.

[43] M. Wachs, M. Schanzenbach, and C. Grothoff. On the feasibility of a censorship resistant decentralized name system. In J. L. Danger, M. Debbabi, J. Marion, J. García-Alfaro, and A. N. Zincir-Heywood, editors, *Foundations and Practice of Security - 6th International Symposium, FPS 2013, La Rochelle, France, October 21-22, 2013, Revised Selected Papers*, volume 8352 of *Lecture Notes in Computer Science*, pages 19–30. Springer, 2013.

[44] A. C. Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164. IEEE Computer Society, 1982.

[45] G. Zhong, I. Goldberg, and U. Hengartner. Louis, lester and pierre: Three protocols for location privacy. In N. Borisov and P. Golle, editors, *Privacy Enhancing Technologies, 7th International Symposium, PET 2007 Ottawa, Canada, June 20-22, 2007, Revised Selected Papers*, volume 4776 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2007.

Appendix

A. MaxPace implementation using BetterTimes-syntax

This section describes how MaxPace can be enforced using BetterTimes-syntax. The concrete protocol is referred to as DecentMP (short for *Decentralized MaxPace*).

A.1. Homomorphic primitives

Recall that for the scope of this paper, Alice holds the private key for all BetterTrees computations and is the only principal able to decrypt data. However, *Bob* is able to perform arithmetic computations using the BetterTrees system. Below, the building blocks needed to construct DecentMP are described (see the result in Listing 3).

The following is constructed using BetterTimes-syntax. All coin tosses can be sampled from a cryptographically secure source using the built-in `random(start, end)` function. By convention, variables storing a ciphertext uses a prefixing "$c_$". As in normal Python, exponentiation is written using double multiplication signs; x^y is written as `x ** y`.

A.1.1. Homomorphic Distance

The squared Euclidean distance (henceforth simply called distance) can be computed using additively homomorphic encryption in a privacy-preserving manner [45,13,40,39,22]. As shown in [21], most approaches are only secure in the semi-honest model but can be made secure in the malicious model using BetterTrees.

The approaches above require that *Bob* holds one of the two coordinates in the clear. Listing 4 shows a short protocol in BetterTimes-syntax syntax which computes the distance between (x_1, y_1) and (x_2, y_2) without any plaintext knowledge. Computing the distance while holding a coordinate in the plain is similar, however where the last two parameters `c_x2, c_y2` are plaintexts and thus have a different type. For the scope of this paper such a method is called $ODist_{plain}$.

Listing 4: Procedure for distance computation

```
def ODist(c_x1, c_y1, c_x2, c_y2):
    c_sq1 = c_x1 * c_x1 + c_y1 * c_y1
    c_sq2 = c_x2 * c_x2 + c_y2 * c_y2
    c_cross = c_x1 * c_x2 + c_y1 * c_y2
    return c_sq1 + c_sq2 - 2 * c_cross
```

A.1.2. Homomorphic Comparisons

There are several solutions to compute comparisons homomorphically in the literature [22,13] by making use of bit parity. Here, a comparison method very similar to the one by Hallgren et al. is used [22].

Hallgren et al. use the fact that $(x - y) \cdot \rho$, with ρ uniformly random, yields $\perp$ if and only if x and y are not equal. Thus, to compare $x < y$, it's possible to check if $\exists i \in \{0..y-1\} : (x - i) \cdot \rho = 0$. However, where Hallgren et al. use an array of equality-checks and shuffle it to hide which slot is equal to the compared value, instead the values are multiplied here, as shown in Listing 5.

Listing 5: Procedure for computing "less than"

```
def lessThan(c_x, y):
    c_l = 1
    for i in range(0, y - 1):
        c_l = c_l * (c_x - i)
    return c_l * random(1, k)
```

A.1.3. Homomorphic Proximity Check

To enforce the MaxPace policy, it is necessary to compute whether two points are near each other. In short, the formula consists of chaining $ODist_{plain}$ and `lessThan`, as shown in Listing 6.

Listing 6: Procedure to check the proximity of two points

```
def proximity(c_x1, c_y1, x2, y2, r):
    dist = ODist_plain(c_x1, c_y1, x2, y2)
    return lessThan(dist, r ** 2)
```

A.1.4. Homomorphic Speed

The following shows, to the best of the authors' knowledge, the first case where speed computations are used together with additively homomorphic encryption. More precisely, *Bob* calculates whether or not the speed of *Alice* is under an allowed threshold, as shown in Listing 7.

Listing 7: Procedure to check for too fast movement

```
def speed(c_x1, c_y1, c_x2, c_y2, h, t):
    dist = ODist(c_x1, c_y1, c_x2, c_y2)
    return lessThan(dist, (h * t) ** 2)
```

The speed when moving d distance over a time t is computed as d/t. In MaxPace, the goal is to check when the speed exceeds a threshold h. Thus, the sought computation is $\frac{d}{t} \leq h$, which can be re-written (for non-negative integers) as $d \leq h \cdot t$.

B. BetterTreesover Rings

Note that additively homomorphic schemes are commonly defined over groups where when multiplying a non zero element γ with a uniformly chosen ρ, the result is not necessarily uniformly distributed, thus potentially affecting the blinding of $g(x, y)$. For instance, in groups such as $\mathbb{Z}_n$ for composite $n = p \cdot q$ (as used by the Paillier [36] encryption scheme) when multiplying a non invertible element with random ρ, the result stays in the subgroup of non-invertible elements. In that setting is thus possible to show a counterexample to the theorem above.

In this section we design a protocol for outsourcing multiplication to Alice over the Ring $\mathbb{Z}_n = \{k \in \mathbb{N} \mid 1 \leq k \leq n\}$ for $n = pq$, where p and q are two different primes. We show that the protocol is secure similar to the protocol over Fields presented in earlier section. Before proceeding to the main protocol we introduce required basics.

For given $n \in \mathbb{N}$, $\mathbb{Z}_n^* = \{k \in \mathbb{N} \mid 1 \leq k \leq n \text{ and } G.C.D(k,n) = 1\}$ denotes the set of all numbers between 1 to n which are co-prime to n. We know that $\mathbb{Z}_n^*$ is a group with respect to multiplication modulo n. The cardinality of $\mathbb{Z}_n^*$ is denoted by $\Phi(n)$. In our case, $\Phi(n)$ is $(p-1)(q-1)$. The number of non co-primes to n are $n - \Phi(n) = pq - (p-1)(q-1) = p + q - 1$. These non-co-primes are essentially multiples of p and q; q of them are multiples of p, p of them are multiples of q and pq is the common multiple. It is easy to see that, if a number $m \in \mathbb{Z}_n$ is not co-prime to n, then $G.C.D(m,n)$ must be either p or q. Therefore, if Bob picks a number m randomly and suppose it is not co-prime to n then Bob can find p (and hence q) *easily* as finding $G.C.D$ is *easy*. As a result, with out loss of generality, we can assume that if Bob picks a number $m \in \mathbb{Z}_n$ then m is most probably co-prime to n.

B.1. Proof sketch

Lemma 4. *In the outsourced multiplication protocol BetterTimesMul the assurance value* $\mathbf{a}$ *is equal to 0 if the protocol is followed, else either the value of* $\mathbf{a}$ *is indistinguishable from* $\perp$ *or Alice learns nothing extra by deviating from the protocol.*

Proof. We first show that if Alice follows the protocol honestly then $\mathbf{a}$ is equal to 0. From the protocol, we have $[a] = \rho[d]$, where $[d] = [a'] - ([z']c_m + [y']c_a)$.

$$\begin{aligned}
[d] &= [a'] - ([z']c_m + [y']c_a) \\
&= [cy'] - ([x'y']c_m + [y']c_a) \\
&= ([x']c_m + [c_a])y' - ([x'y']c_m + [y']c_a) \\
&= ([x'y']c_m + [y']c_a) - ([x'y']c_m + [y']c_a) \\
&= [0]
\end{aligned} \tag{12}$$

This implies, $[a] = \rho[d] = [\rho d] = [0]$.

Now we proceed to show that if Alice deviates from the protocol then either the value of $\mathbf{a}$ is indistinguishable from $\perp$ or Alice learns nothing extra by deviating from the protocol.

It is easy to see that, if $d \in \mathbb{Z}_n^*$ then $\mathbf{a}$ is indistinguishable from $\perp$ as ρd is random when $\rho \neq 0$ and d is invertible. This implies, at the same time if Alice wants to cheat Bob and distinguish ρd from $\perp$ then she must choose $d \in \mathbb{Z}_n \setminus \mathbb{Z}_n^*$. That is, $G.C.D(d,n)$ must be either p or q. Without loss of generality assume that $G.C.D(d,n) = p$ and Alice choose d such that $d = kp$ for some $k \in \{1, 2, \ldots, q-1\}$. Recall that, $[d] = [a'] - ([z']c_m + [y']c_a)$.

$$\begin{aligned}
[d] &= [a'] - ([z']c_m + [y']c_a) \\
[kp] &= [a'] - ([z']c_m + [y']c_a) \\
[a'] &= ([z']c_m + [y']c_a) + [kp] \\
[a'] &= [z'c_m + y'c_a + kp].
\end{aligned} \tag{13}$$

We notice that, Alice can always compute $[\alpha] = [a'] - [c]y' - [k]p$ as she knows a', c, y', k and p.

$$\begin{aligned}
[\alpha] &= [a'] - [c]y' - [k]p \\
&= [a'] - ([x']c_m + [c_a])y'] - [kp] \\
&= [a'] - ([x'y'c_m + y'c_a + kp]) \\
&= [a' - (x'y'c_m + y'c_a + kp)] \\
&= [z'c_m + y'c_a + kp - (x'y'c_m + y'c_a + kp)] \\
&= [(z' - x'y')c_m].
\end{aligned} \tag{14}$$

This shows that Alice can always compute $\alpha = (z' - x'y')c_m$. If $(z' - x'y')$ is invertible then Alice can compute $c_m = \alpha(z' - x'y')^{-1}$ as well. But, the probability that the Alice can *guess* c_m from c is $\frac{1}{n}$. Therefore, if Alice deviates from the protocol then with high probability $(z' - x'y')$ is non-invertible, i.e., $(z' - x'y') \in \mathbb{Z}_n \setminus \mathbb{Z}_n^*$.

In any case, the goal of Alice is to return $[z']$ which is different from $[x'y']$ such that $d \in \mathbb{Z}_n \setminus \mathbb{Z}_n^*$ and $(z' - x'y') \in \mathbb{Z}_n \setminus \mathbb{Z}_n^*$. As $(z' - x'y') \in \mathbb{Z}_n \setminus \mathbb{Z}_n^*$, $G.C.D(z' - x'y', n)$ must be either p or q but not pq, since $z' \neq x'y'$. With out loss of generality, assume that $G.C.D(z' - x'y', n) = q$. This implies, $z' - x'y' = lq$ for some l, where $1 \leq l \leq p - 1$. This implies, $z' = x'y' + lq$ and $[z'] = [x'y'] + [lq]$.

If Alice cheats, then she receives the final value $[Res + \rho d + lqw]$ from Bob, for some $w \in \mathbb{Z}_n$ depending on the circuit. $[Res + \rho d + lqw]$ = $[Res + \rho kp + lqw]$ since $d = kp$ for some k. □

B.2. Insecure Functionalities over Rings

Some functionalities may be insecure over rings, even if they are evaluated with BetterTrees. One such example is the InnerCircle protocol by Hallgren et al. [22]. InnerCircle is a protocol for deciding whether two principals are closer to each other than some threshold r. This is accomplished using a distance computation as shown in Section 4.1.1, followed by a novel comparison technique. Hallgren et al. prove the protocol secure over fields in the semi-honest model, however even though the distance computation can be made secure in the malicious model (as explained in Section 4.1.1), the comparison technique can not. This is due to the fact that the functionality assumes that multiplication with any non-zero element by a random number yields another number, which to the attacker is indistinguishable from random (it is $\perp$). However, using for instance Paillier, we have that any multiplication with a product of p is another product of p (and likewise for q). The comparison method proceeds as shown in Listing 8.

Listing 8: Procedure for computing “less than"

```
def lessThan(c_x, y):
    l = []
    for i in range(0, y - 1):
        l.append((c_x - i) * random(1, k))
    return l
```

The idea is that the list `l` contains only random numbers if $x \geq y$, or only random numbers except for exactly one zero if $x < y$. However, if there are subgroups (like with Paillier), this will also leak whether

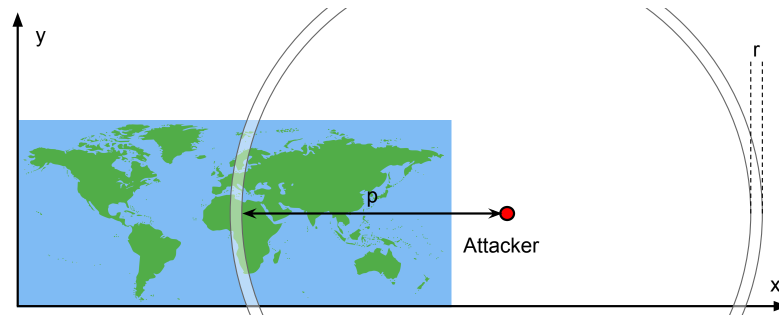

Fig. 10. Visualized attack on the InnerCircle protocol

or not $x \geq t \cdot p \vee x < t \cdot p + r$ for some $t \in \{0..q\}$ (and vice versa for q). It may seem far fetched that a number is close to the secret factors p or q, but in the case of InnerCircle, x is the distance which means that the adversary can make this fact likely by positioning themselves at a large distance away from the victim.

For InnerCircle, this leakage is also rather meaningful. Assuming that the plane is large (e.g. with Paillier, each axis would be at least 2^{1024}), the coordinates needed to cover the earth are in least-significant corner. If the attacker positions them far from earth, e.g. p units from the center of the earth's surface, then the `lessThan` function will compute not the proximity of the attacker and the victim, but whether the victim is a band of width r, spanning the height of the earth. For an example, see Figure 10, where the attacker, positioned at the red dot, learns if the victim is in an area stretching from Cape Town in the south of Africa to Svalbard north of Europe, covering several entire European countries such as Italy and Sweden.