Maturity and Performance of Programmable Secure Computation*

David W. Archer
dwa@galois.com

Dan Bogdanov
dan.bogdanov@cyber.ee

Benny Pinkas
benny@pinkas.net

Pille Pullonen
pille.pullonen@cyber.ee

October 27, 2016

1 Introduction

Secure computation (SC) stands for a group of technologies for computing functions of private inputs, while keeping the inputs themselves hidden. The canonical example of secure computation is the millionaires’ problem, where two millionaires, Alice and Bob, who own $X$ and $Y$, respectively, wish to run a computation that tells them which one of them is richer, but reveals no other information. Obviously, if both parties trust some third party they could reveal $X$ and $Y$ to that party, who could then tell them whether $X > Y$. Their goal is, however, to do the same computation without the help of any third party and while revealing nothing more than the final output of the computation.

Secure computation is essentially based on processing data that is protected by encryption or a similar method. There are SC solutions that are targeted for computing specific functions. We call a particular secure computation technology programmable if it is Turing-complete and can efficiently run at least a certain class of algorithms. Most SC solutions are designed in order to protect data during sharing or outsourced processing, for example in the context of cloud computing. Technologies for programmable SC include (but are not limited to) secure multi-party computation (MPC) and fully homomorphic encryption (FHE).

Secure computation research has gained traction internationally in the last five years. In the United States, the DARPA PROCEED program (2011-2015) focused on development of multiple SC paradigms and improving their performance. In the European Union, the PRACTICE program (2013-2016) focuses on its use to secure cloud computing. Both programs have demonstrated exceptional prototypes and performance improvements.

∗This is an extended version of [ABPP16]
In PROCEED, Archer and Rohloff [AR15] demonstrated VoIP streaming where an untrusted server decompresses, mixes, adds, clips, and recompresses audio data while it remains encrypted. Carter et al. demonstrated the capability to compute route maps while map, source, and destination remain secret [CLT14a]. In PRACTICE, Bogdanov et al. evaluated a tax fraud detection system together with the Estonian Tax and Customs Board [BJSY15] and showed significant speedups of MPC using cloud computing. In addition, both programs contributed to speeding up the basic technologies of SC.

In this paper, we collect the results from both programs and other published literature to present the state of the art in what can be achieved with today’s secure computing technology. In the following, Sec. 2 describes three approaches of programmable secure computation that we analyse in this paper. These are homomorphic encryption, garbled circuits and linear secret sharing. This introduction is followed by a set of interesting properties in Sec. 3 that are used to characterize and differentiate between the approaches. In Sec. 4 we present a taxonomy based on implementation maturity and runtime performance of each technique showcasing the readiness for real-world use. The taxonomy has five components—usage model, programming paradigm, implementation maturity, developer tool maturity and performance. Sec. 5 gives concrete performance evaluation of different secure computation artifacts based on common benchmark applications like AES evaluation. Finally, Sec. 6 summarises collected information using the proposed maturity taxonomy.

## 2 Secure Computation Paradigms

Secure computation is a multi-party processing of private data where different parties play different roles. The computing parties $C$ are the ones actually carrying out the computations. The input parties $I$ give their private data for the computations, and it is important to ensure that the data remains private except for the desired computation outcomes. Finally, the outcomes are obtained by the result parties $R$. One participant may carry several of these roles, for example a party that gives inputs and receives outputs is denoted as $IR$. The theory of secure computation is mostly centred around the computing parties and often expects the result and input parties to be the same as the computing parties (depicted as $ICR$ on Fig. 1c). However, many practical deployments, such as surveys, separate these roles. A longer discussion about roles and deployment scenarios can be found in [BKLPV13] and in Sec. 4.

Secure computation can be done in many ways depending on the needed functionality and existing resources. In general, SC is required if it is necessary to avoid leaking any information except for the final output of the computations that is given to the result parties. A secure computation protocol can be designed to have passive security, also known as security against semi-honest adversaries, meaning that it is secure if the computing parties follow the protocol but might try to infer extra information from what they see during the protocol. A protocol secure against actively malicious participants is secure even if the
computing parties try to cheat and do not follow the protocol. The intermediate case, security against covert adversaries, guarantees that cheating participants are caught with a reasonable probability, say 25%. This security guarantee is effective if the participants have a strong incentive not to be caught cheating, but might try to deceive if it is likely to avoid detection. Security against actively malicious adversaries is stronger than security against covert adversaries, which in turn offers more guarantees than passive security.

Most of the secure computation literature handles the case of corrupted computation parties, and gives absolute freedom for the input players to choose their inputs, and for the output players to choose the functionality that is computed. However, a corrupt behaviour of these parties can easily render the computation useless or insecure. For example, if the result parties are allowed to propose queries or algorithms for the computation then such corrupted parties might try to learn more outputs than originally intended. In this case, the computing parties should verify that it is safe to run each piece of the computation. On the other hand, input parties might try to corrupt the computation by giving invalid inputs. To that end, the computing parties should obliviously verify that the inputs fall to the desired bounds or have the desired format and discard invalid inputs and outliers from the computation. Furthermore, the input and result parties should have means to check that the outputs are correct and the computation really followed the safe procedure.

There are several major technologies that are used for secure computation, on which we elaborate in the next sections. In case there is one well equipped server, fully homomorphic encryption (FHE) can be deployed. Two-party computation is well supported by garbled circuits (GC) as well as linear secret sharing (LSS). However, the latter also allows for secure multi-party computation, involving more than two parties (GC can also be applied in the multi-party setting [BMR90, BNP08], but most of the research on GC focuses on the two-party setting). We note that there are additional technologies for secure computation, but they have received less attention both in the theoretical literature and with developers. For example, the Random Access Machine (RAM) model using oblivious RAM constructions, e.g. ObliVM [LWN+15].

2.1 Homomorphic Encryption

Homomorphic encryption (HE) is an encryption scheme that enables computations on encrypted values. Figuratively, HE is an opaque locked glovebox [Gen09a]. A party can input its valuables to the box and lock it. Anyone with the box can use the gloves to manipulate the items inside, but only the box owner has the key to open the box and take the contents out. Hence, HE is an encryption with means to combine ciphertexts so that the result is a meaningful operation like addition or multiplication on plaintexts. These operations can be applied to the ciphertexts even without knowing the decryption key. Commonly HE is considered in a two-party setting where the client (input and result party) has the keys and outsources some computation to the server (the computing party) by providing it with the encrypted inputs. Such division of the roles is
represented by Fig. 1a. At its best, HE requires interaction only for sending the inputs and retrieving the outputs, making it very communication efficient.

Many schemes allow to compute one kind of operation over encrypted data, for example addition in the Paillier encryption scheme \cite{Pai99} or multiplication in the Elgamal \cite{Elg85} scheme. Somewhat homomorphic encryption (SWHE) allows for computing both operations, but only a limited amount of one of them. Usually operations introduce noise to the ciphertext and after some operations the level of noise is too high for successful decryption. However, starting with seminal work of Gentry \cite{Gen09a, Gen09b}, fully homomorphic encryption (FHE) schemes that allow for unlimited number of both operations have become feasible. Remarkably, these two operations enable to compute any arithmetic functionality. In general, FHE is achieved from SWHE by introducing bootstrapping phase that resets the noise to a low level. FHE schemes used in secure computation include DGHV \cite{VDGHV10}, NTRU \cite{HPS98} and BGV \cite{BGV12}. As a practical example, NTRU has been used for secure teleconferencing and e-mail filtering \cite{AR15}.

2.2 Garbled Circuits

The first secure computation method was Yao’s GC \cite{Yao82,LP09,BHR12} proposed in 1982. Garbled circuits are like integrated digital circuits where it is hard to observe the values carried between single gates and only the output of the total circuit is revealed. Moreover, the materials used in the construction are fragile and dissolve after one use. Hence, although the circuit diagram remains the same a new circuit needs to be built for every evaluation.

In more detail, the idea is to evaluate boolean circuits by encoding wire values as random strings and encrypting the truth tables of each gate. The encodings of the input wires of a gate can be used to decrypt the encoding corresponding to the gate output. The first party, the garbler, chooses the encodings, generates the encrypted truth tables and forwards the circuit to the evaluator. The other party, the evaluator, obtains the encodings corresponding to the secret inputs and uses them to decode the encrypted truth tables and obtain the output. Both parties have the role of the computing party and usually both also provide inputs and obtain the outputs as on Fig. 1c however this may vary. Furthermore the process can be modified to allow external input and result parties. In recent years there has been tremendous improvement in the performance of GC protocols, resulting in many efficient prototypes such as justgarble \cite{BHKR13}, SCAPI \cite{EFLL12}, and tinygarble \cite{SHS15}. The basic method is secure against passive adversaries, however, recent research mostly considers active security \cite{LP07,NO09,LP11,Lin13,MR13}.

The GC approach excels in high latency networks as it requires a small number of rounds of interaction. It is straightforward to securely compute any functionality as there exist several compilers (e.g. \cite{HKV12,KMsB13}) that produce optimized circuits and software libraries implementing GC computation. On the downside, many interesting functionalities have huge boolean circuits and thus require a lot of bandwidth to transfer the garbled circuit. Actively secure GC is
deployed, for example, by Dyadic Security for safeguarding cryptographic keys from corrupt administrators by splitting them between several servers and computing encryption without storing the keys in a single location [Sec14]. Another example of GC deployment is finding common contacts between Android users in application CommonContacts[1].

2.3 Linear Secret Sharing

Secret sharing was proposed in 1979 [Sha79, Bla79] and is the basis for a prolific branch in secure computation with seminal works [GMW87, BGW88, CCD88]. To illustrate LSS-based computation, consider a set of interconnected gloveboxes with input hatches. Any party can distribute their valuables between the boxes through the hatches. Afterwards the box operators can exchange pieces and manipulate the inputs using the gloves. However, the final product is obtained only when all boxes are opened and their results are combined.

More concretely, LSS enables parties to divide secrets to multiple shares where any unqualified set of shares does not reveal information about the secret. Each share is given to a different party. Homomorphic properties of the sharing are used to apply arithmetic operations without revealing the shared values. Each arithmetic operation is computed collaboratively by a dedicated interactive protocol that is run between the computing parties and large functionalities can be combined from basic operations or have a new specialized protocols. The strength of LSS is allowing reactive protocols where new inputs depend on the previous outputs, as well as securely storing intermediate results. The computing parties carry out the interactive protocols, however especially in the case of passive security it is straightforward to incorporate external input and result parties as on Fig. 1b.

A significant advancement of SC was LSS-based computation deployment for the Danish sugar beet auction [BCD09] in 2008. Current practical implementations of LSS-based secure computation include Sharemind [BLW08, Bog13] and ShareMonad [LDDAM12] for passive security and SPDZ [DPSZ12, DKL13] and TinyOT [NNOB12, LOS14] for active security. ShareMonad is used for spam filtering and secure teleconference [LADM14, AR15]. Actively secure LSS is deployed to secret share cryptographic keys and compute cryptographic operations using secure computation thereby mitigating threats from corrupted servers [Sec14]. Sharemind has been deployed to analyse ICT companies economic indicators [BTW12], perform genome-wide association studies [KBLV13], run government statistics [Kam15] and detect tax fraud [BJSV15].

---

3 Security Properties and Comparison Criteria

The main goal of SC is to enable useful and potentially collaborative computations while hiding the private data of the input parties. A protocol is considered secure if the only thing revealed in the computation is the output (and, of course, whatever information that can be deduced from the output). Although all SC protocols follow this general definition, the settings in which they are proposed differ significantly. This section mentions important theoretical criteria that can be used to label SC protocols as well as points out common properties of different paradigms where possible. We base this section on a recent classification by Perry et al [PGFW14]. In the following, Sec. 4 considers a classification from a more practical perspective.

As introduced in Sec. 2, two important criteria for characterizing secure computation are the computation technology, where we consider FHE, GC and LSS, and the adversarial model which is either active, passive or covert. Characteristics that are tightly coupled with the computation paradigm are the model of computation and the number of communication rounds. Common computation models include boolean and arithmetic circuits, although other models such as random access machines and Turing machines are also occasionally used. The garbled circuits method is mostly described for boolean circuits whereas LSS is applied to arithmetic circuits (although both methods can also be applied to the other kind of circuits). FHE schemes support either one of the circuit models depending of the concrete scheme. GC and FHE have a constant number of communication rounds while the number of communication rounds of LSS schemes is linear in the depth of the circuit that is computed.

Different schemes can also be compared based on the desired deployment scenario. A central property is the number of computing parties required for the computation coupled with the fraction of tolerated corrupted parties for which the protocol is still secure. FHE and GC focus on two party computation where one party is allowed to be corrupted. LSS works for any number of parties but a common model used in practice is two or three computing parties with one corrupted party. A significant exception is the SPDZ model that allows to corrupt all but one of the participants.

The communication model can assume that all parties have point-to-point connections or that there exists a broadcast channel (a broadcast channel is relevant in the case that there are more than two parties). All considered schemes except SPDZ work in a point-to-point setting. SPDZ requires a broadcast channel but also discusses the possibility of obtaining broadcast via a specific protocol in a point-to-point setting. In addition, it is often meaningful to divide computation to a preprocessing phase that does not require any knowledge of the actual inputs and an online phase that uses preprocessing results and the actual inputs to efficiently compute the result. This setting is applicable when it is known beforehand that some computations are coming up. Out of the aforementioned schemes, TinyOT and SPDZ use preprocessing.
From a programming perspective it is important to consider handling conditional statements. Conditional statements with secret conditions are commonly processed by evaluating all branches and obliviously choosing the right outcome. Many properties describe the behaviour of the adversary or the possible outcomes for the corrupted parties.

- Commonly the model of corruption is static meaning that corrupted parties are fixed ahead of the protocol, but it is also possible to consider adaptive adversaries that decide during the protocol execution which parties to corrupt.

- A protocol is fair if whenever an output is obtained all parties are guaranteed to receive it (rather than the adversary being able to obtain the output while keeping other participants from learning it).

- An abort capability means that a protocol run can be interrupted without leaking information about the input. The reconstruction capability means that it is possible for honest parties to restore the output even if corrupted parties stop participating in the protocol. Schemes considered in this paper are capable of abort but not of reconstruction.

- Interesting but more theoretical properties include the security assumptions and whether security is preserved in a concurrent execution. For example, security can be information theoretic meaning that it is infeasible for any adversary (even with infinite computation powers) to break the scheme. Alternatively, security can be based on computational assumptions, meaning that breaking security in reasonable time is equivalent to breaking some well known hardness assumption (such as the hardness of factoring large numbers). Information theoretic security can only be achieved if a majority of the parties are honest. Passively secure LSS-based multi-party computation usually has information theoretic security whereas all two-party computation schemes offer only computational security.

- The security level estimates the expected number of operations required to break the scheme (say, by doing a brute-force search over all possible keys). The level of security is measured in bits where $b$-bit security means that the attack is expected to take $2^b$ operations. It is customary to support at least 80 bit or 128 bit of security.

Most of the research on SC focuses on the computing parties. The input or result parties could also be computing parties or could alternatively outsource the computation. In particular, it is often reasonable to consider settings where the computing parties are some fixed entities to whom input parties can send private data and from whom result parties can request queries. In this context, it is also possible to ask which guarantees regarding the correctness or privacy of the computation can be given to the input and result parties that do not take part in the computation. For example, these parties may be able
to verify the correctness of the result or audit the computation process. In theory public verification is possible for all SC protocols, however doing so efficiently is currently an open question. Auditing the computation process has been studied for Sharemind [Pik14] and verification of outputs has been studied for SPDZ [BD014]. FHE-based SC is also well suited for outsourcing and achieving verifiability although no practical auditing solutions have currently been implemented.

4 Maturity Taxonomy

This section provides a taxonomy for secure computation techniques that aims to summarize various aspects of real world use of SC. We consider five different features — the usage model, programming paradigm, implementation maturity, developer tool maturity and performance. This section focuses on aspects of practical usability complementing the formal characterization criteria from Sec. 3.

\[ \text{TR} \xleftarrow{\text{SC}} \xrightarrow{\text{C}} \text{I} \xrightarrow{\text{SC}} \xleftarrow{\text{R}} \xleftarrow{\text{ICR}} \xrightarrow{\text{SC}} \]

(a) Outsourced processing   (b) Outsourced services   (c) Joint processing

Figure 1: Party roles and communication in abstract usage models of SC

Not all secure computation techniques are well-suited for all kinds of applications. We define three general usage models that describe how data is obtained and used by the application. Each of these is illustrated in Table 1 by well-known services that could be replaced with analogous privacy preserving tools. The separation of the expected roles of the parties together with the direction of communication is illustrated by Fig. 1. Out of the considered secure computation techniques, HE is well suited for outsourced processing whereas LSS and GC are better for outsourced services and joint computations. A developer considering the use of secure computation should also be aware of possible programming paradigms for a chosen method. Either the programmer designs boolean or arithmetic circuits or is able to write programs that will be interpreted by the computation framework. The programming paradigms for different secure computation techniques are collected in Table 2.

Different SC paradigms are represented by various concrete protocol sets and frameworks for secure computation. We describe these using an implementation maturity category in Table 3 together with a Technological Readiness Level (TRL) \(^\text{2}\). This category indicates the engineering level of the best publicly known

### Table 1: Usage model for secure computing systems

<table>
<thead>
<tr>
<th>Category</th>
<th>Criteria of belonging</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Outsourced processing</td>
<td>A client uses external resources (e.g., a cloud) to process its own data, and seeks protection against resource controller.</td>
<td>Salesforce, Erply</td>
</tr>
<tr>
<td>Outsourced services</td>
<td>A client uses external resources (e.g., a cloud) to process data collected from multiple data owners, while protecting this data from the resource controller and from itself.</td>
<td>Google Forms, Survey-Monkey.</td>
</tr>
<tr>
<td>Joint processing</td>
<td>Multiple clients collaborate to process data collected from among themselves, protecting their own data from each other.</td>
<td>Tinder, Doodle</td>
</tr>
</tbody>
</table>

### Table 2: Programming paradigm used by secure computing systems

<table>
<thead>
<tr>
<th>Category</th>
<th>Criteria of belonging</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circuits</td>
<td>Task expressed as a fully formed boolean or arithmetic circuit.</td>
<td>Yao-style GC, TinyOT</td>
</tr>
<tr>
<td>Programs</td>
<td>Task expressed as a continuously interpreted program of arbitrarily complex primitive operations.</td>
<td>LSS-based SC</td>
</tr>
</tbody>
</table>

Table 1: Usage model for secure computing systems

Table 2: Programming paradigm used by secure computing systems
implementations of different paradigms. Currently, LSS is the most used secure computation method, however GC approaches are also featured in market-ready and real-world deployment levels. Similarly, we describe the developer tool maturity in Table 4 to show which tools are available for developing applications with different SC frameworks. Most systems propose a special language that can be used to program applications but some also provide various levels of libraries to ease the development.

<table>
<thead>
<tr>
<th>Level</th>
<th>Criteria of belonging</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Academic prototype</td>
<td>Implementation demonstrated in a laboratory setting (TRL 1-4).</td>
<td>Fairplay (GC) [MNPS04, BNP08], HElib (FHE) [HS14], SEPIA (LSS) [BSMD10], TASTY (GC &amp; HE) [HKS+10], ShareMonad (LSS), VIFF (LSS) [DGKN09]</td>
</tr>
<tr>
<td>Real-world deployment</td>
<td>Implementation demonstrated in a real-world setting (TRL 5-6).</td>
<td>FastGC (GC) [HEKM11], FRESCO (LSS) [Han15]</td>
</tr>
<tr>
<td>Market-ready</td>
<td>Commercial services available based on technology (TRL 7-9).</td>
<td>Dyadic (GC &amp; LSS), Partisia (LSS), Sharemind (LSS)</td>
</tr>
</tbody>
</table>

Table 3: Implementation maturity of secure computation systems

<table>
<thead>
<tr>
<th>Level</th>
<th>Criteria of belonging</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming library</td>
<td>A hand-modified implementation or a library of primitives for integration.</td>
<td>FRESCO, HElib, ShareMonad, SEPIA, VIFF</td>
</tr>
<tr>
<td>Domain-specific language</td>
<td>An embedded or compilable domain-specific language targeted to secure computing.</td>
<td>LI [SKB+09, SKM11], OblivC [ZEL5], PCF [KMsB13], SecreC [Jag10, BLR14], SFDL [MNPS04, BNP08]</td>
</tr>
<tr>
<td>High-level libraries &amp; tools</td>
<td>Reusable application-specific functionalities or tool integrations.</td>
<td>SecreC standard library, Rmind [BKLS14]</td>
</tr>
</tbody>
</table>

Table 4: Developer tool maturity of secure computation systems

There are many secure computation frameworks proposed in the literature, but not all of them have practical implementations in software or hardware. The classification in Table 5 assigns the performance level category to secure computation frameworks to denote the performance level of more mature implemen-
tations available in practice. In the following, Sec. 5 compares the performance of different systems in more depth based on concrete benchmarks.

<table>
<thead>
<tr>
<th><strong>Level</strong></th>
<th><strong>Criteria of belonging</strong></th>
<th><strong>Examples</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-operation-level</td>
<td>Performed primitive operations on input arrays of non-trivial size.</td>
<td>HElib [GHJR14]</td>
</tr>
<tr>
<td>Algorithm-level</td>
<td>Runs an algorithm built of multiple primitive operations on an input structure of non-trivial size.</td>
<td>FRESCO [DDN+15], SEPIA [BD11, MBD12]</td>
</tr>
<tr>
<td>Business-process-level</td>
<td>Runs a multi-algorithm business process on an input database of non-trivial size.</td>
<td>Sharemind [Kam15, Sec. 6.4]</td>
</tr>
</tbody>
</table>

Table 5: Performance levels of secure computing systems

5 Performance of SC Implementations

A comprehensive survey performed near the end of the DARPA PROCEED program characterized the performance of the three SC paradigms described in Sec. 2. We describe salient results of that survey here. First, we aim to provide a pragmatic answer to the question, "How fast is FHE anyway?" using a number of benchmarks. Second, we aim to compare all three SC paradigms using the comparison benchmark of computing the AES-128 cipher.

5.1 How Fast is FHE?

Table 6 shows results from several publications on FHE performance for basic algorithms. The artifacts referenced here were all created using fully homomorphic (FHE) techniques. In each case, the artifact is tolerant of passive adversaries. Results show a wide range of performance for the same operation computed using different FHE approaches, as shown in the first two table entries. Other results in this table show that the relative performance for different artifacts using the same FHE method is sometimes non-intuitive. For example, the difference of 3 orders of magnitude between 32-bit addition and multiplication is not analogous to typical computation results “in the clear”.

Although details are somewhat non-intuitive, the table offers an intuitive feel for current FHE performance. This and other data gathered during the PROCEED program suggest that as of early 2015, FHE computation is often between 5 and 10 orders of magnitude slower than computing “in the clear”.
### 5.2 Cross-paradigm SC Performance on AES-128

The computation of the AES cipher (represented as a circuit) is a convenient benchmark for comparing the performance of SC techniques. For example, Damgård et al. report on a pre-2012 comparison of several GC and linear secret sharing systems, comparing them by AES performance. In Fig. 2, we report on amortized cost for block evaluation results from the 2012-2015 time frame. Across a range of 9-bit to 128-bit security levels, performance results span almost seven orders of magnitude, from roughly 50 microseconds per block to roughly 200 seconds per block. Note that some of these results are based on running multiple threads in parallel or need a preprocessing computation which is not included in the online runtime.

LSS implementations tend to achieve the highest throughputs shown on the chart, ranging from roughly 50 microseconds to 0.5 seconds per block. Results of Sharemind \cite{LTW13, Cyb15} and TinyOT \cite{NNOB12} also demonstrate that increased amortization gives efficiency gains, especially for TinyOT this is even true when increasing the security parameter. GC implementations cover the middle range of performance shown, ranging from roughly 20 milliseconds per block to 60 seconds per block. FHE implementations demonstrate the slowest throughputs in the time frame of our comparison, ranging from roughly 8 seconds per block to 200 seconds per block. The best pictured amortized cost of secure AES is about two orders of magnitude slower than non-amortized software AES.
Figure 2: Comparison of SC paradigms using the AES-128 block cipher
5.3 Levenshtein Distance Computation

Edit distance is a sought after primitive in the processing genomic data. Therefore, Levenshtein algorithm was useful as a SC benchmark in the PROCEED program, though we found no other published results on this benchmark. In Fig. 3 we compare performance results as a function of input string length for linear secret sharing and garbled circuit constructions. In the chart, the red datapoints represent a ShareMonad (LSS) solution developed by Galois, Inc., the blue data points represent a Sharemind (LSS) solution developed by Cybernetica, and the green points represent a GC solution developed by a team at University of Oregon and Georgia Tech. Measurements were made on a 2.1GHz Intel e7 (Xeon) CPU with 1TB physical memory. The consistency of the results is surprising, given the variability of FHE-specific data and of the AES comparisons discussed above.

5.4 SC in Complex Applications

Applications more complex than simple operations or algorithms are difficult to compare across SC paradigms. Partly, because the choice of underlying algorithms can vary substantially between implementations, since SC is typically not the only measurable contributor to performance in such applications. In ad-
dition, user expectations of performance of complex applications are typically subjective rather than absolute. Thus in this paper we list complex applications that have been successfully implemented in SC with “reasonable” performance, but do not report on specific performance measurements of those implementations. In Table 7 we show for each considered SC paradigm a selection of complex applications that have been implemented.

<table>
<thead>
<tr>
<th>LSS</th>
<th>GC</th>
<th>FHE</th>
</tr>
</thead>
<tbody>
<tr>
<td>VoIP with encrypted server signal processing</td>
<td>VoIP with encrypted server voice addition</td>
<td>Forensic image recognition</td>
</tr>
<tr>
<td>E-mail filtering using regular expressions</td>
<td>E-mail filtering using string match only</td>
<td>Genetic association study algorithms</td>
</tr>
<tr>
<td>Naive Bayes spam filtering</td>
<td>Linear best-fit regression</td>
<td>Linear programming based credit ranking</td>
</tr>
<tr>
<td>Satellite collision analysis</td>
<td>SHA2-based web service authentication</td>
<td>Route mapping</td>
</tr>
<tr>
<td>Genome SNP correlation to medical conditions</td>
<td>Tax fraud detection</td>
<td>Fingerprint identification</td>
</tr>
<tr>
<td>Private statistics to analyse ICT students dropping out and working during studies</td>
<td>Private statistics to analyse ICT students dropping out and working during studies</td>
<td>Finding similar patients</td>
</tr>
<tr>
<td>Linear programming based credit ranking</td>
<td>Linear programming based credit ranking</td>
<td>Finding similar patients</td>
</tr>
</tbody>
</table>

Table 7: Complex SC applications implemented with reasonable performance

6 Maturity of Secure Computation

We collected information on the performance results and published prototypes and used it to estimate the maturity of programmable secure computation techniques. Table 8 evaluates the five most popular technologies with their current state of the art estimations. The schemes that achieve practical efficiency are currently mostly passively secure. Hence, currently secure computation is a good approach for tasks with multiple parties who trust each other to behave honestly but still need to deploy means to ensure the privacy of the computations. However, ongoing research and development is largely focused on active security and it is likely to gain more efficiency in the near future.

It is still not clear, which fields will benefit the most from programmable secure computation or whether it will find industrial acceptance. However, continued research in the field as well as increasingly larger real-world deployments suggest that anyone looking for privacy-preserving computing technology keep an eye on the development of secure computation.
<table>
<thead>
<tr>
<th>Technique</th>
<th>Implementation maturity</th>
<th>Programs</th>
<th>Developer tool maturity</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSS passive</td>
<td>Outsourced services / Joint processing</td>
<td>Market-ready</td>
<td>High-level libraries &amp; tools</td>
<td>Business-process level</td>
</tr>
<tr>
<td>LSS active</td>
<td>Outsourced services / Joint processing</td>
<td>Market-ready</td>
<td>Programming library</td>
<td>Algorithm-level</td>
</tr>
<tr>
<td>GC passive</td>
<td>Outsourced services / Joint processing</td>
<td>Real-world deployment</td>
<td>Domain-specific language</td>
<td>Business-process level</td>
</tr>
<tr>
<td>GC active</td>
<td>Outsourced services / Joint processing</td>
<td>Market-ready</td>
<td>Circuits</td>
<td>Algorithm-level</td>
</tr>
<tr>
<td>FHE passive</td>
<td>Outsourced processing / Outsourced services</td>
<td>Academic prototype</td>
<td>Programming library</td>
<td>Single-operation level</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Usage model</th>
<th>Implementation maturity</th>
<th>Programs</th>
<th>Developer tool maturity</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSS passive</td>
<td>Outsourced services / Joint processing</td>
<td>Market-ready</td>
<td>High-level libraries &amp; tools</td>
<td>Business-process level</td>
</tr>
<tr>
<td>LSS active</td>
<td>Outsourced services / Joint processing</td>
<td>Market-ready</td>
<td>Programming library</td>
<td>Algorithm-level</td>
</tr>
<tr>
<td>GC passive</td>
<td>Outsourced services / Joint processing</td>
<td>Real-world deployment</td>
<td>Domain-specific language</td>
<td>Business-process level</td>
</tr>
<tr>
<td>GC active</td>
<td>Outsourced services / Joint processing</td>
<td>Market-ready</td>
<td>Circuits</td>
<td>Algorithm-level</td>
</tr>
<tr>
<td>FHE passive</td>
<td>Outsourced processing / Outsourced services</td>
<td>Academic prototype</td>
<td>Programming library</td>
<td>Single-operation level</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Technique</th>
<th>Usage model</th>
<th>Implementation maturity</th>
<th>Programs</th>
<th>Developer tool maturity</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSS passive</td>
<td>Outsourced services / Joint processing</td>
<td>Market-ready</td>
<td>Programs</td>
<td>High-level libraries &amp; tools</td>
<td>Business-process level</td>
</tr>
<tr>
<td>LSS active</td>
<td>Outsourced services / Joint processing</td>
<td>Market-ready</td>
<td>Programming library</td>
<td>Algorithm-level</td>
<td></td>
</tr>
<tr>
<td>GC passive</td>
<td>Outsourced services / Joint processing</td>
<td>Real-world deployment</td>
<td>Circuits</td>
<td>Domain-specific language</td>
<td>Business-process level</td>
</tr>
<tr>
<td>GC active</td>
<td>Outsourced services / Joint processing</td>
<td>Market-ready</td>
<td>Circuits</td>
<td>Domain-specific language</td>
<td>Algorithm-level</td>
</tr>
<tr>
<td>FHE passive</td>
<td>Outsourced processing / Outsourced services</td>
<td>Academic prototype</td>
<td>Circuits</td>
<td>Programming library</td>
<td>Single-operation level</td>
</tr>
</tbody>
</table>

Table 8: Maturity of most popular programmable SC techniques

---

1. Partisia Market Design ApS: [www.partisia.dk](http://www.partisia.dk) / Cybernetica AS: [sharemind.cyber.ee](http://sharemind.cyber.ee/)  
3. Sugar beet auction: [BCD+09](http://alexandra.dk/uk/cases/simap)  
4. Economic indicators: [BTW12](http://alexandra.dk/uk/cases/simap), tax fraud detection: [BJSV15](http://alexandra.dk/uk/cases/simap), government statistics: [Kam15](http://alexandra.dk/uk/cases/simap)  
6. Simplex algorithm for linear programming: [DDN+15](http://www.mightbeevil.com/framework/)  
7. Dijkstra’s shortest path algorithm on PCF: [github.com/cryptouva/pcf](https://github.com/cryptouva/pcf) and White-wash: [CLT14a](http://www.mightbeevil.com/contacts/)
References


