It's Not Just a Phase: Understanding Function Dependence in MPC Preprocessing

Andreas Brüggemann
brueggemann@encrypto.cs.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Kasra Edalatnejad
edalat@encrypto.cs.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Jan Filipp
filipp@encrypto.cs.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Philipp Kusterer
philipp.kusterer@stud.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Thomas Schneider
schneider@encrypto.cs.tu-darmstadt.de
Technical University of Darmstadt
Darmstadt, Germany

Abstract

Secure multiparty computation (MPC) is a key enabling technology for privacy-preserving applications. To achieve low latency after inputs are provided, many efficient protocols utilize preprocessing, and the landscape of preprocessing approaches is growing and diverse. Some recent works use variants of *function-dependent* preprocessing, leveraging additional knowledge on the function to be computed for even better performance. While the existing literature primarily focuses on performance, it is mostly overlooked that there exist different kinds of *function dependence* and what practical implications they have. We systematize the different flavors of *function dependence*, relating them to practical requirements beyond efficiency.

CCS Concepts

• **Security and privacy → Cryptography**.

Keywords

Multiparty Computation; Preprocessing; Function Dependence

ACM Reference Format:
Andreas Brüggemann, Kasra Edalatnejad, Jan Filipp, Philipp Kusterer, and Thomas Schneider. 2026. It's Not Just a Phase: Understanding Function Dependence in MPC Preprocessing. In *25th Workshop on Privacy in the Electronic Society (WPES '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3847192.3847357

1 Introduction

Secure multiparty computation (MPC) is a cryptographic tool that enables mutually distrusting parties to jointly compute on their private data without disclosing any information but the desired result. MPC receives increasing interest from industry (for an overview see [22, 26]), providing the opportunity to compute on sensitive data while safeguarding its privacy. One popular practical goal is to minimize the latency from parties providing their inputs to them learning their outputs. To achieve that, it is common to move input-independent computation to a preprocessing phase, preceding the online phase where the parties provide their actual inputs. Early but still crucial examples include the precomputation of Beaver triples [2] to accelerate later multiplications, or oblivious transfer (OT) precomputation [3]. Advancements in MPC research have yielded more specific kinds of preprocessing, including preprocessing for more specialized operations such as lookup tables (LUTs) [6, 17], matrix multiplication [8, 44], and shuffling [10, 38], and some approaches now even leverage additional knowledge of specific circuit topologies to further boost efficiency [4, 49]. Preprocessing that depends on the function to be computed has been coined *function-dependent preprocessing*, e.g. [14, 48, 56], and it has also been criticized, e.g., for a lack of flexibility [5, 38] and practical efficiency concerns [33]. Yet, function dependence is not a binary property: preprocessing may depend on an entire circuit topology or only on an upper bound of the circuit size, but it may also depend on what exactly is computed by LUTs within a circuit, dimensions of matrices to be multiplied, number of values to be shuffled, etc. Although there exist some approaches also using a more fine-grained terminology like distinguishing between general purpose and special functions [53], or separating phases that depend on the function-size and -topology [24], those works only consider single aspects of preprocessing function dependence.

In this work, we investigate and make the first steps towards a common terminology for different kinds and degrees of function dependence and independence in preprocessing present in the literature, and we shed light on practical considerations of preprocessing, such as batching, flexibility, and storage of preprocessing material.

2 Different Flavors of Preprocessing

We begin by separating certain one-time operations from the main preprocessing, and then review different kinds of function dependence in the main preprocessing. Then, we review further properties of function dependence and their implications.

2.1 One-Time Setup

Certain preprocessing needs to be done only once, irrespective of the exact function to be computed (or its representation's size),

This work is licensed under a Creative Commons Attribution 4.0 International License.
WPES '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3026-9/2026/11
https://doi.org/10.1145/3847192.3847357

Table 1: Overview of various preprocessing approaches for different operations.

Operation	Style	Material	Topology Dependence	Parameter Dependence
Multiplication $x \cdot y = z$ / AND $x \wedge y = z$	Beaver [2, 13, 16, ...]	$([a], [b], [c = ab])$	—	—
	ABY2.0 [4, 11, 31, 39, 49, ...]	$([\lambda_x], [\lambda_y], [\gamma_{xy} = \lambda_x \lambda_y])$	local	—
	DN07 [12, 21, ...]	$([r]_{\text{deg}}, [r]_{2\cdot\text{deg}})$	—	—
	Yao/GC [36, 57, 58]	$\left\{E_{k_x^i, k_y^j}\left(k_{z^{i \wedge j}}\right)\right\}_{i,j \in \{0,1\}}$	global	—
Matrix Mult. $\vec{X} \cdot \vec{Y} = \vec{Z}$	Matrix triples [44]	$([\vec{A}], [\vec{B}], [\vec{C} = \vec{A}\vec{B}])$	—	matrix dim.
	Vectorized ABY2.0 [8, 49]	$([\vec{\lambda_X}], [\vec{\lambda_Y}], [\vec{\gamma_{XY}} = \vec{\lambda_X}\vec{\lambda_Y}])$	local	matrix dim.
LUT $F(\vec{x}) = \vec{y}$	rotated LUT/OTTT [17, 28]	$[F'(x) = F(x \oplus r)], [r]$	—	$F, \#in, \#out$
	FLUTE [6]	$\left([\lambda_{x_i}]\right)_{1 \le i \le \#in}, \left([\gamma_I = \prod_{i \in I} \lambda_{x_i}]\right)_{I \in 2^{\{1,\ldots,\#in\}}}$	local	$\#in$
	rand. one-hot vector [45, 55]	$([\vec{e_r}], [r])$	—	$\#in$
	Yao/GC [43, 57]	$\left\{E_{k_{x_1}^{i_1}, \ldots, k_{x_{\#in}}^{i_{\#in}}}\left(k_{y_j}^{F(\vec{x})_j}\right)\right\}_{i_1,\ldots,i_{\#in} \in \{0,1\}; 1 \le j \le \#out}$	global	$F, \#in, \#out$
Shuffle $\pi(\vec{x}) = \vec{y}$	Share translation [10]	$P_0 : \pi, \pi(\vec{a}) + \vec{b}; \quad P_1 :$ rand. $\vec{a}, \vec{b}$	—	$\#in = \#out$
	Shared perm.+corrections [38]	$P_0 : \vec{r_0}, \pi_0, \pi_0', \pi_1(\vec{r_1}) - \vec{s}; \quad P_1 : \ldots$	—	$\#in = \#out$
	Permutation-matrix [7]	$[\vec{M_\pi}]$ + ABY2.0-style preproc.	local	$\#in = \#out$
Conversion arith.↔binary	daBits [52]	$([r]_b, [r]_a)$	—	—
	edaBits [20]	$([r_0]_b, \ldots [r_{k-1}]_b, \left[\sum_i r_i 2^i\right]_a)$	—	domain size k

and it may be usable over many, independent protocol executions for different functions. Popular examples of that in MPC protocols include the exchange of PRF seeds to enable subsets of parties to later non-interactively sample the same random values, e.g., [1, 39], or running a low number of random base OTs linear in the security parameter that are extended to polynomially many OTs via OT extension [27]. One may even count operations that are on a lower level as part of the preprocessing. For instance, when secure channels are needed, their setup may require additional preparation such as setting up certificates, key agreement, etc.

We call this kind of preprocessing *one-time setup* here, as it is done only once in the very beginning before running many instances of MPC protocols. It is independent of the function(s) to be computed, and the resulting constant cost usually is quite low, so that it is insignificant compared to the remaining cost incurred by running the MPC protocol or, if used, any further preprocessing that is not one-time. Hence, in the remainder of this work, we focus on the other kinds of preprocessing that we identify in the literature.

2.2 Main Per-Gate Preprocessing

To identify different kinds of function dependence (or independence) in MPC protocols, we review preprocessing techniques for popular building blocks (gates) in MPC protocols in the literature. We emphasize that we do not provide a full review of the vast MPC literature. Rather, we identify and discuss a selection of techniques that serves as a cross section through the MPC literature, featuring different notions of function dependence and providing examples for their sometimes subtle differences.

In Table 1, we compare different styles of preprocessing for various operations, which we elaborate on in the following. For our comparison, we focus on what preprocessing material is generated on a per-gate basis while omitting some inexpensive local operations like invocations of a PRF for the sake of clarity. Regarding function dependence, we point out dependence on the topology of the circuit representing the function to be computed, as well as dependence on parameters such as the number of inputs $\#in$ or number of outputs $\#out$ to an operation.

The most prominent operations in MPC are multiplications or their binary variant, AND gates. A well-known example for preprocessing is the generation of **Beaver triples** [2] $([a], [b], [c])$ in some additively linear secret sharing $[\cdot]$, where $c = ab$ and a, b are uniformly random. One such triple later enables to efficiently multiply two values x, y online, and the preprocessing is independent of the function to be computed. Another approach is **DN07**-style multiplication [12] which per multiplication precomputes a double sharing of the same random element $[r]$ encoded in Shamir-style polynomials of degree deg and $2 \cdot$ deg without any function dependence.

This is contrasted by **masked secret sharing** techniques as in ABY2.0 [49] that cut the online communication of multiplication with Beaver triples in half. Here, each wire holds a linear secret sharing of a mask $[\lambda_v]$ chosen during preprocessing, and in the online phase, each party learns the secret value offset by the mask λ_v. For a multiplication gate, input wire masks $[\lambda_x], [\lambda_y]$ are multiplied during preprocessing, yielding the product $[\gamma_{xy}]$. Hence, the preprocessing material can also be seen as a triple $([\lambda_x], [\lambda_y], [\gamma_{xy}])$. Yet, in contrast to Beaver triples [2], masks $[\lambda_x], [\lambda_y]$ are not uniformly random, but linear combinations of random masks sampled for outputs of prior multiplications and inputs, depending on the circuit topology between these wires and the multiplication of x and y. Hence, they can be correlated, while the $[a], [b]$ of different Beaver triples are always independent. As an example, for multiplications $v \cdot w$ and $v \cdot (w + x)$, one needs correlated

triples $([\lambda_v], [\lambda_w], [\gamma_{vw}])$ and $([\lambda_v], [\lambda_y := \lambda_w + \lambda_x], [\gamma_{vy}])$. Thus, this approach is commonly called *function-dependent preprocessing* in the literature. Here, we state more precisely that the preprocessing is topology-dependent, and we call it *locally topology-dependent* as it only depends on the topology of some subcircuit. For a multiplication on masked secret sharings, this is the subcircuit between the multiplication and other interactive gates feeding this multiplication, including linear operations in between.

Finally, the **garbling** approach [57] enables to garble AND gates during preprocessing, depending on keys of the input and output wires. Hence, the preprocessing of one gate transitively depends on that of all previous gates, so that this can be considered a *global topology dependence*, which is a stronger dependence than the previously discussed local topology dependence in, e.g., ABY2.0 [49].

Beyond simple scalar multiplications and AND gates, there also exist more specialized operations such as **matrix multiplication** (or dot products, as a special case), or **lookup tables (LUT)**, where a LUT represents an arbitrary function $F(\vec{x}) = \vec{y}$. Their preprocessing might now also depend on parameters like the size of gates, e.g. matrix dimension or LUT input and output sizes. This dependence, which we call *parameter dependence*, again is a spectrum. On one end there are matrix multiplication protocols like [8, 44, 49], that are based on Beaver triples [2] and scalar ABY2.0 [49], generalized from scalars to matrices, and directly depending on the matrix dimension of the multiplication's input. The LUT preprocessing approaches [17, 28] compute a sharing of the LUT permuted by a random offset $[r]$, but require the exact number of LUT inputs, outputs, and the represented function F to be known in advance. Towards the other side of the spectrum are protocols with low parameter dependence, like FLUTE [6] and [55], that only depends on the number of inputs, but not on the number of outputs nor the function F itself. When comparing [17, 28] (high parameter-dependence; topology independence) and FLUTE [6] (low parameter-dependence; local topology dependence), then for both the preprocessing is somewhat function dependent but with vastly different flavors. Similar observations can be made, e.g., for **shuffling** and **conversions**.

Finding 1. *Function dependence is not a binary property, but a spectrum along at least two dimensions: 1) (Circuit)* topology dependence *might not exist (e.g., Beaver multiplication [2]), might be* local *to specific subcircuits (e.g., ABY2.0 [49]), or might be* global *(e.g., garbling [57]). 2) For specialized operations, there can be* parameter dependence*, e.g., w.r.t. concrete input and output dimensions, orthogonal to potential topology dependence.*

2.3 Further Observations and Considerations

Having looked at topology and parameter dependence, we proceed with miscellaneous further properties of function dependence, and their practical implications and considerations.

Pooling of Preprocessing Material. Even function-independent preprocessing has some dependence on the function: the number of gates that require preprocessing, e.g., the number of AND gates. Yet, e.g., Beaver triples [2] provide even more flexibility. They are not necessarily linked to a specific protocol execution, so it not only suffices to know an upper bound of preprocessed gates in advance, but excess preprocessing material can simply be used for future protocol runs. Yet, this freedom is not present for all such protocols, especially for active security. E.g., [5] runs a verification over a certain number of AND gates at the end of the protocol. This verification requires preprocessing, parametrized only by the number of AND gates (or an upper bound), but linking specific Beaver triples to the proof, so that these cannot simply be used in another execution if they remain unused. If, e.g., Beaver triples include MACs as in SPDZ [13], preprocessing material is bound to a MAC key, limiting transferability between protocol executions. Using the same MAC key for multiple executions can be possible, but brings some pitfalls, like that upon failure of a MAC check, all triples using the same MAC key need to be discarded [40].

Finding 2. *Some preprocessing approaches provide the flexibility of computing pools of preprocessing material to be dynamically consumed across different protocol executions as needed, but even function-independent preprocessing may lack this flexibility.*

Batching. Preprocessing enables to create large batches of the same type of preprocessing material like multiplication triples. This batching can improve the per-unit cost, e.g., by exploiting vectorization like SIMD, AVX, or vectorized AES [46], and especially in the malicious setting where techniques such as cut-and-choose [41] are used. While batching works best for low function dependence (e.g., each Beaver triple [2] is generated exactly the same way), batching may also be possible for function-dependent approaches such as garbled circuits. For example, works such as LEGO [47, 48] create garbled gates in large batches to have better cost aggregation for malicious checks via cut-and-choose. DUPLO [35] may generate sub-circuits for common functionalities like applying AES in large batches and use them as a commodity. Also, topology-dependent preprocessing as in ABY2.0 [49] can be efficiently instantiated from Beaver triples as shown in [6], again enabling better batching.

Finding 3. *Preprocessing enables to parallelize similar operations in batches for practical performance improvements and amortization, and function dependence does not preclude batching.*

"Hidden" Parameters. Some parameters are often considered to be parameters for the protocol, e.g., domain size and security parameters. Hence, they are not listed in §2.2 as a per-gate dependence. Still, e.g., the domain size can be variable, even within the same protocol execution [50], so it is also a kind of function dependence.

Finding 4. *MPC parameters such as domain sizes are often assumed given but can be critical parameters to the preprocessing. Often, preprocessing material can only be used for one specific domain size.*

Modifiability. The lower the function dependence of preprocessing, the more freedom one has to decide what function(s) to compute after the preprocessing. Yet, even preprocessing with strong function dependence allows later modifications of the function. E.g., while the garbled circuit approach is globally topology dependent, LEGO [47, 48] and DUPLO [35] garble single gates or smaller subcircuits while then allowing to arbitrarily *solder* them together at some additional cost. For example, one may precompute critical building blocks like AES with higher function dependence and efficiency and then solder them into larger circuits as needed [35]. Regarding the secret-sharing-based protocols with local topology dependence from Table 1, we discover that *soldering* is also possible. This can be done by interactively resharing wires from one to another preprocessing mask at the soldering point, using the down- and upgrade

procedures from [9]. As the function dependence is only local, one even gains flexibility without adding interaction. The output of an interactive gate (multiplication, LUT, etc.) can be soldered to any unused wire for free, because the output mask is independent of the gate inputs and the gate itself. Furthermore, NOT-gates and additions by public values can be arbitrarily added, as they do not change the masks and hence have no influence on the preprocessing. Finally, flexible gates such as LUTs can be reprogrammed after preprocessing, e.g., in FLUTE [6], given that the preprocessing has no parameter dependence on the represented function.

Finding 5. *Function-dependent preprocessing can still maintain some flexibility out-of-the-box. Running function-dependent preprocessing for smaller subcircuits and then* soldering *them together as needed can provide both, efficiency benefits of function-dependent preprocessing and flexibility, enabling more fine-grained tradeoffs.*

Ad Hoc Preprocessing. The natural approach to preprocessing is to run it in advance before the online phase. While the preprocessing impacts the cost of running the system, its latency (i.e., waiting for network transfer) does not matter, as it does not involve any user waiting for the output. However, in practice, solutions may not always have this clean separation between the preprocessing and the online phase and instead run ad hoc preprocessing pipelined with the online phase, e.g., MP-SPDZ [32], TASTY [23], and FastGC [25]. Such overlap of preprocessing and online phase has practical implications: 1) Both the function and input are known during preprocessing, s.t. not knowing the function is no longer a limitation. 2) The latency of preprocessing impacts the latency of the online phase. 3) Since the communication costs also overlap, any solution that reduces online communication at the price of higher total communication is no longer beneficial. 4) Preprocessing material needs to be stored for a shorter time, improving storage complexity. This makes many of the masked secret sharing approaches [11, 39, 49] less suitable for overlapping phases. While this ad hoc approach technically is no *pre*-processing, it may still enable to improve performance by better batching (cf. Finding 3) or creating pools of preprocessed material (cf. Finding 2) that are replenished as needed.

Finding 6. *Preprocessing can be done ad hoc instead of in advance, but this has substantially different practical implications.*

Storage Complexity. Running preprocessing before the online phase requires parties to store the preprocessed material until it is consumed. These materials may 1) be derived from short PRF seeds or PCGs [5, 15, 51], e.g., random masks in [11, 39], or 2) be transmitted and stored, e.g., the preprocessed products of aforementioned random masks. While it is possible to expand short seeds during the preprocessing phase [33], postponing the PRF evaluations to the online phase can significantly reduce the storage requirement and loading cost, while PRF evaluation with fast hardware-backed AES calls does not have a significant impact on online latency. Beyond the choice of early expansion, the storage medium creates another storage-latency tradeoff with essentially two options: 1) *RAM*, which is very efficient but limited and expensive, while keeping large amounts of preprocessing material in RAM is not practical when the two phases are far apart. 2) *Disk*, which is cheap and usually available in high volume, but loading the material from disk to RAM before using it increases latency.

Assuming that the data is already loaded in RAM is not an uncommon simplifying assumption in evaluation. This assumption omits the cost of loading a potentially high volume of material into RAM when preprocessing does not overlap with the online phase. In an outsourcing setting where the material does not depend on the client's identity, it is possible to keep a hot storage loaded ahead of time [19]; however, when materials are client-specific [30], preloading them ahead of time without knowing who the next client would be is not possible, which exacerbates the problem.

Finding 7. *For practical efficiency, it is crucial how preprocessing material is stored. Works relying on preprocessing should clearly state the storage complexity (besides, e.g., run time, communication), and benchmarks should account for the cost of loading material from disk.*

Preprocessing with a Dealer. Many works consider a trusted dealer that inexpensively provides the (online) parties with preprocessing material [5, 11, 38, 51, 55]. Here, a low level of function dependence can be beneficial. A dealer can provide preprocessing material "as a service", and keeping this material as generic as possible makes the dealer more generic, making it easier to deploy by some third party, maybe, e.g., even on trusted hardware [15, 29]. Also, low function dependence means that less information about the kind of function(s) that are computed needs to be leaked to the dealer.

Finding 8. *Lower function dependence may render it more practical to utilize preprocessing material provided by a dealer party.*

Functions vs Inputs. The boundary between what is function vs. what is input is not always clear. For example, when performing polynomial evaluation $P(x) = y$, the polynomial may be considered as an input to the function, e.g., a party inputting the polynomial roots α_i and setting the function as $F(x, \{\alpha_i\}) = \prod_{i\in[d]}(x - \alpha_i)$, versus when the polynomial is the function to be evaluated, e.g., computing $F_{a_0,\ldots,a_d}(x) = \sum_{i\in[d]} a_i x^i$ where the a_is are the coefficients of the polynomial. While these two represent the same ideal functionality, they differ in "function dependence". While an input can be considered as a function argument, the other direction is also possible as universal circuits (UC) [54] and private function evaluation (PFE) [18, 34, 37, 42] encode the function to be evaluated as part of the input and can make any preprocessing function independent.

Finding 9. *A clear distinction between functions and inputs is crucial to discussing function dependence in preprocessing, as the same information may be encoded as either a function or an input, depending on the application.*

3 Conclusion & Call to Action

This paper highlights that there are more nuances to preprocessing than merely classifying each approach as either *function dependent* or *function independent*, and there are many practical aspects to be taken into consideration to decide which kind of preprocessing fits the requirements of an application best.

We recommend future MPC works relying on preprocessing to make statements regarding function dependence more precise, e.g., by distinguishing between different kinds of parameter and topology dependence. Furthermore, we recommend to clearly report both online *and* preprocessing complexities (e.g., run time and communication), as well as address further metrics like storage complexity.

Acknowledgments

This project received funding from the ERC under the EU's research and innovation program Horizon Europe (PRIVTOOLS/101124778). It was co-funded by the DFG within SFB 1119 CROSSING/236615297, and supported by BMFTR and HMWK within ATHENE.

References

[1] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *ACM CCS*.
[2] Donald Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In *CRYPTO*.
[3] Donald Beaver. 1995. Precomputing Oblivious Transfer. In *CRYPTO*.
[4] Aner Ben-Efraim, Michael Nielsen, and Eran Omri. 2019. Turbospeedz: Double Your Online SPDZ! Improving SPDZ Using Function Dependent Preprocessing. In *ACNS*.
[5] Elette Boyle, Niv Gilboa, Matan Hamilis, Yuval Ishai, and Ariel Nof. 2025. Preprocessing for Life: Dishonest-Majority MPC with a Trusted or Untrusted Dealer. In *IEEE S&P*.
[6] Andreas Brüggemann, Robin Hundt, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2023. FLUTE: Fast and Secure Lookup Table Evaluations. In *IEEE S&P*.
[7] Andreas Brüggemann, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2022. Poster: Efficient Three-Party Shuffling Using Precomputation. In *ACM CCS*.
[8] Andreas Brüggemann, Oliver Schick, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2024. Don't Eject the Impostor: Fast Three-Party Computation With a Known Cheater. In *IEEE S&P*.
[9] Andreas Brüggemann, Thomas Schneider, and Maximilian Stillger. 2027. Additions, Multiplications, and the Interaction In-Between: Optimizing MPC Protocols via Leveled Linear Secret Sharing. In *NDSS*.
[10] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. 2020. Secret-Shared Shuffle. In *ASIACRYPT*.
[11] Harsh Chaudhari, Ashish Choudhury, Arpita Patra, and Ajith Suresh. 2019. ASTRA: High Throughput 3PC over Rings with Application to Secure Prediction. In *ACM CCSW@CCS*.
[12] Ivan Damgård and Jesper Buus Nielsen. 2007. Scalable and Unconditionally Secure Multiparty Computation. In *CRYPTO*.
[13] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. 2012. Multiparty Computation from Somewhat Homomorphic Encryption. In *CRYPTO*.
[14] Ivan Damgård and Sarah Zakarias. 2013. Constant-Overhead Secure Computation of Boolean Circuits using Preprocessing. In *TCC*.
[15] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2014. Ad-Hoc Secure Two-Party Computation on Mobile Devices using Hardware Tokens. In *USENIX Security*.
[16] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*.
[17] Ghada Dessouky, Farinaz Koushanfar, Ahmad-Reza Sadeghi, Thomas Schneider, Shaza Zeitouni, and Michael Zohner. 2017. Pushing the Communication Barrier in Secure Computation using Lookup Tables. In *NDSS*.
[18] Yann Disser, Daniel Günther, Thomas Schneider, Maximilian Stillger, Arthur Wigandt, and Hossein Yalame. 2023. Breaking the Size Barrier: Universal Circuits Meet Lookup Tables. In *ASIACRYPT*.
[19] Kasra Edalatnejad, Sebastian Faust, Jonas Hofmann, Philipp-Florens Lehwalder, and Thomas Schneider. 2026. Do You Need a Receipt? Anonymous Credential Revocation at Continental Scale via Private Record Certification. In *USENIX Security*.
[20] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. 2020. Improved Primitives for MPC over Mixed Arithmetic-Binary Circuits. In *CRYPTO*.
[21] Vipul Goyal, Hanjun Li, Rafail Ostrovsky, Antigoni Polychroniadou, and Yifan Song. 2021. ATLAS: Efficient and Scalable MPC in the Honest Majority Setting. In *CRYPTO*.
[22] UC Berkeley EECS Security Group. 2026. MPC Deployments. https://mpc.cs.berkeley.edu/ (accessed 11.09.2026).
[23] Wilko Henecka, Stefan K ögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg. 2010. TASTY: Tool for Automating Secure Two-partY computations. In *ACM CCS*.
[24] Marco Holz, Ágnes Kiss, Deevashwer Rathee, and Thomas Schneider. 2020. Linear-Complexity Private Function Evaluation is Practical. In *ESORICS*.
[25] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. 2011. Faster Secure Two-Party Computation Using Garbled Circuits. In *USENIX Security*.
[26] MPC Alliance Inc. 2026. https://www.mpcalliance.org (accessed 11.09.2026).
[27] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending Oblivious Transfers Efficiently. In *CRYPTO*.
[28] Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, Claudio Orlandi, and Anat Paskin-Cherniavsky. 2013. On the Power of Correlated Randomness in Secure Computation. In *TCC*.
[29] Kimmo Järvinen, Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. 2010. Embedded SFE: Offloading Server and Network using Hardware Tokens. In *Financial Crypto*.
[30] Neha Jawalkar, Kanav Gupta, Arkaprava Basu, Nishanth Chandran, Divya Gupta, and Rahul Sharma. 2024. Orca: FSS-based Secure Training and Inference with GPUs. In *IEEE S&P*.
[31] Banashri Karmakar, Nishat Koti, Arpita Patra, Sikhar Patranabis, Protik Paul, and Divya Ravi. 2024. Asterisk: Super-fast MPC with a Friend. In *IEEE S&P*.
[32] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *ACM CCS*.
[33] Marcel Keller. 2025. Rep3 Reloaded: On the Cost of Function-Dependent Preprocessing in Semi-Honest 3PC with Honest Majority. Cryptology ePrint Archive, Paper 2025/919. https://eprint.iacr.org/2025/919
[34] Ágnes Kiss and Thomas Schneider. 2016. Valiant's Universal Circuit is Practical. In *EUROCRYPT*.
[35] Vladimir Kolesnikov, Jesper Buus Nielsen, Mike Rosulek, Ni Trieu, and Roberto Trifiletti. 2017. DUPLO: Unifying Cut-and-Choose for Garbled Circuits. In *ACM CCS*.
[36] Vladimir Kolesnikov and Thomas Schneider. 2008. Improved Garbled Circuit: Free XOR Gates and Applications. In *ICALP*.
[37] Vladimir Kolesnikov and Thomas Schneider. 2008. A Practical Universal Circuit Construction and Secure Evaluation of Private Functions. In *Financial Crypto*.
[38] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. 2024. Graphiti: Secure Graph Computation Made More Scalable. In *ACM CCS*.
[39] Nishat Koti, Mahak Pancholi, Arpita Patra, and Ajith Suresh. 2021. SWIFT: Super-fast and Robust Privacy-Preserving Machine Learning. In *USENIX Security*.
[40] Alexander Kyster, Frederik Huss Nielsen, Sabine Oechsner, and Peter Scholl. 2025. Rushing at SPDZ: On the Practical Security of Malicious MPC Implementations. In *IEEE S&P*.
[41] Yehuda Lindell and Benny Pinkas. 2007. An Efficient Protocol for Secure Two-Party Computation in the Presence of Malicious Adversaries. In *EUROCRYPT*.
[42] Hanlin Liu, Yu Yu, Shuoyao Zhao, Jiang Zhang, Wenling Liu, and Zhenkai Hu. 2021. Pushing the Limits of Valiant's Universal Circuits: Simpler, Tighter and More Compact. In *CRYPTO*.
[43] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella. 2004. Fairplay—A Secure Two-Party Computation System. In *USENIX Security*.
[44] Payman Mohassel and Yupeng Zhang. 2017. SecureML: A System for Scalable Privacy-Preserving Machine Learning. In *IEEE S&P*.
[45] Hiraku Morita, Erik Pohle, Kunihiko Sadakane, Peter Scholl, Kazunari Tozawa, and Daniel Tschudi. 2025. MAESTRO: Multi-Party AES Using Lookup Tables. In *USENIX Security*.
[46] Jean-Pierre Münch, Thomas Schneider, and Hossein Yalame. 2021. VASA: Vector AES Instructions for Security Applications. In *ACSAC*.
[47] Jesper Buus Nielsen and Claudio Orlandi. 2009. LEGO for Two-Party Secure Computation. In *TCC*.
[48] Jesper Buus Nielsen, Thomas Schneider, and Roberto Trifiletti. 2017. Constant Round Maliciously Secure 2PC with Function-independent Preprocessing using LEGO. In *NDSS*.
[49] Arpita Patra, Thomas Schneider, Ajith Suresh, and Hossein Yalame. 2021. ABY2.0: Improved Mixed-Protocol Secure Two-Party Computation. In *USENIX Security*.
[50] Deevashwer Rathee, Mayank Rathee, Rahul Kranti Kiran Goli, Divya Gupta, Rahul Sharma, Nishanth Chandran, and Aseem Rastogi. 2021. SiRnn: A Math Library for Secure RNN Inference. In *IEEE S&P*.
[51] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. 2018. Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications. In *ASIACCS*.
[52] Dragos Rotaru and Tim Wood. 2019. MArBled Circuits: Mixing Arithmetic and Boolean Circuits with Active Security. In *INDOCRYPT*.
[53] Shuang Sun and Eleftheria Makri. 2025. SoK: Multiparty Computation in the Preprocessing Model. Cryptology ePrint Archive, Paper 2025/060. https://eprint.iacr.org/2025/060
[54] Leslie G. Valiant. 1976. Universal Circuits (Preliminary Report). In *STOC*.
[55] Sameer Wagh. 2022. Pika: Secure Computation using Function Secret Sharing over Rings. *PoPETs* (2022).
[56] Xiao Wang, Samuel Ranellucci, and Jonathan Katz. 2017. Authenticated Garbling and Efficient Maliciously Secure Two-Party Computation. In *ACM CCS*.
[57] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *SFCS*.
[58] Samee Zahur, Mike Rosulek, and David Evans. 2015. Two Halves Make a Whole. In *EUROCRYPT*.