An Embedded Domain-Specific Language for Logical Circuit Descriptions with Applications to Garbled Circuits

Andrei Lapets Wyatt Howe Ben Getchell Frederick Jansen
Nth Party, Ltd.
Boston, MA
{andrei, ben, wyatt, frederick}@nthparty.com

Abstract

Contemporary libraries and frameworks that make it possible to incorporate secure multi-party computation protocols and capabilities into production software systems and applications must sometimes deliver underlying capabilities (such as logical circuit synthesis) to new kinds of environments (such as web browsers or serverless cloud computing platforms). In order to illustrate some of the benefits of addressing this challenge by building a solution from the ground up that leverages the features of a contemporary and widely used programming language, we present an embedded domain-specific language that allows programmers to describe and synthesize logical circuits. Notably, this approach allows programmers to employ many of the language features and any of the programming paradigms supported by the host language. We illustrate this flexibility by considering two use cases: synthesizing circuits for relational operations and synthesizing circuits corresponding to the SHA-256 cryptographic hash function.

1. Introduction

Software applications, services, and systems that employ secure multi-party computation (MPC) [15, 18] can allow organizations and individuals to offer and utilize data workflows and web services that can operate on encrypted data without decrypting it. This makes it possible to enjoy the benefits of workflows and services while reducing or eliminating the risks and liabilities associated with sharing or storing sensitive data [3, 12].

Some secure MPC frameworks and libraries employ garbled circuit protocols [7, 19] in order to allow participating parties to perform computations using sensitive data that must not be shared. By design, these protocols allow the evaluation of *logical circuits* on the private input data. This means that in order to evaluate an algorithm for which only a high-level description exists, the algorithm must be compiled at some point into a logical circuit. This logical circuit can then be used by the framework as it steps through the protocol, making it possible to evaluate the function the circuit represents.

One traditional approach to implementing software solutions that employ garbled circuit protocols is to split the development effort into two stages. In the first stage, circuits are synthesized offline for low-level building blocks (such as arithmetic operations). In the second stage, these building blocks are composed as necessary to achieve the algorithm's functionality (typically by the application developer, potentially with the aid of an interpreter or compiler). Within both of these stages, it is possible to use traditional circuit description languages and synthesis tools, including well-established solutions such as VHDL and Verilog [8] that are widely adopted throughout the industry. However, this approach has a few drawbacks.

First, many algorithms deployed to address real-world use cases must operate on inputs of different sizes. Even cryptographic primitives such as hash functions may have internal components that run for a number of iterations that depends on the input size. Synthesizing a circuit for every possible input size may be inefficient (*e.g.*, requiring extra storage for each circuit variant) or even infeasible (*e.g.*, if the range of expected input sizes is large). The ability to synthesize circuits in real time could address this challenge.

Second, traditional languages and tools for encoding circuit descriptions and performing circuit synthesis are not always well-suited for use within contemporary software applications. As secure MPC techniques quickly mature and are incorporated into software systems and applications that are deployed in production [12, 14], it is increasingly the case that MPC frameworks and libraries must (1) support rapid integration of MPC protocols into applications built using contemporary software stacks that rely on languages such as Python and JavaScript and (2) be compatible with mobile operating systems, web browsers, and cloud-based environments such as Amazon Lambda and Google Cloud Functions.

Third, it may be easier for a new developer, client, or auditor to evaluate the performance or to verify the correctness of the high-level implementation of an algorithm using a language that is familiar to them. Examining an equivalent circuit description written in a specialized hardware description language or a corresponding synthesized circuit represented as a netlist may be cumbersome or infeasible.

To address these drawbacks, we propose an approach that relies on the development of an *embedded domain-specific language* (EDSL) [5] for building circuit descriptions and performing circuit synthesis. Such a language can be an effective tool within at least two types of scenarios (which are *not* mutually exclusive) involving secure MPC protocols that rely on garbled circuits. In the first type of scenario, specific variants of circuits (built using specific ensembles of gates that are suitable for particular protocols) are hand-crafted by a programmer using features, libraries, and paradigms found in a programming language with which they are familiar. These circuits can then be incorporated into the overall application. By using an EDSL within a familiar host language, a programmer can leverage the full power of that language to build circuit descriptions. In the second type of scenario, variants of circuits are synthesized (based on abstract templates or patterns) on-demand within secure MPC workflows. The availability of a well-designed EDSL may mean that abstract circuit descriptions can be implemented that are similar *or identical* to the high-level functions the programmer must define. Furthermore, because circuit descriptions *are* functions, they can be evaluated in real time to produce circuit variants (based on the sizes of the inputs, the particular collection of logical gates supported by an MPC framework, and other parameters) during the operation of the application.

2. Background and Related Work

Efforts related to the topic of this work can be separated into at least two categories: (1) research related to embedded domain-specific languages for circuit descriptions, and (2) research related to programming tools and frameworks that support the assembly of software solutions that employ secure multi-party computation.

There exists a rich ecosystem of mature frameworks and domain-specific languages for describing and designing hardware (including logical circuits) that have many features and associated tools [16]. Examples include Verilog and VHDL [8], as well as embedded hardware description languages [2]. Such tools are well-suited for assembling circuit descriptions and for synthesizing optimized circuit implementations. However, integrating such tools into real-world software applications (especially applications deployed in specific kinds of environments such as web browsers or serverless computing platforms) may lead to cumbersome compatibility issues. Such tools are also not always easily integrated into applications that must synthesize circuits in real time.

Of some relevance are low-level and high-level circuit description languages and associated synthesis tools for specific domains such as quantum computing [6]. These framework and language design efforts are similar to those being undertaken by the secure MPC community (and includes past efforts [10, 11] by some of the co-authors of this work), as in both cases there is a need for a pipeline that translates high-level descriptions of algorithms into *abstract* logical circuits (*i.e.*, logical circuits that will not be *directly* implemented as physical hardware but will be used as a set of instructions to conduct some sequence of procedures).

There is now an extensive and diverse array of frameworks that can be used when assembling software solutions that employ MPC [9]. This includes frameworks that support contemporary software stacks and environments [1], each relying on either a single protocol or multiple protocols. Some frameworks [13, 17, 20] employ garbled circuit protocols [7]. Employing these protocols to enable the execution of a high-level algorithm on private inputs requires the synthesis of logical circuits that correspond to that algorithm. Such systems sometimes rely on collections of pre-built circuits or specialized synthesis algorithms (*e.g.*, for common arithmetic or cryptographic operations). A strict separation between circuit synthesis and the high-level aspects of a use case may limit opportunities for optimizations (such as refactoring) that depend on context.

3. Embedded Domain-Specific Language for Circuit Descriptions

The EDSL presented in this work allows programmers to assemble logical circuit descriptions and to synthesize logical circuits while still utilizing the rich and expansive array of supported programming paradigms, language features, built-in libraries, and third-party packages of the Python host language. This is accomplished by leveraging for the implementation of the EDSL several important features of Python: higher-order functions, inheritance, operator overloading, decorators, and type annotations.

A core design principle of the EDSL is that programmers (or other automated or semi-automated tools) using the host language to describe a circuit *already know how to define a circuit as a function*. Thus, any EDSL that allows programmers to build up circuits should (1) minimize the number of idiosyncratic patterns and amount of boilerplate necessary to create a circuit description and at the same time (2) maximize compatibility with the language's imperative, functional, and object-oriented programming features.

The EDSL consists of two open-source Python packages. The *circuit*[1] package provides a data structure for representing circuits. Instances of this data structure are the outputs of the circuit synthesis process (and can be converted into other popular representations such as the Bristol Fashion format[2] using Python packages such as *bfcl*[3]). The *circuitry*[4] library includes two class definitions: one for individual bits and one for bit vectors. In their simplest form, these objects behave just like bits and bit vectors: they can be instantiated to specific values (such as `0` or `1`) and then Python infix operators and other methods can operate on them (this is accomplished via operator overloading). However, these objects can also be *abstract*, representing not specific values but entire *circuits* that may evaluate to *many* values. Thus, a Python expression such as `x & y` may simultaneously represent (when evaluated) a concrete value such as `1` and an entire circuit consisting of an output logical conjunction gate whose two inputs are the outputs of the circuits `x` and `y`.

To further aid programmers, the circuitry library provides decorators that can be applied to definitions of functions that operate on bits and bit vectors. These decorators can perform circuit synthesis or can extend the definition of the function so that it can be applied either to a bit vector or to a vector of input gates. When the function is given a vector of input gates and evaluated using the Python interpreter, it synthesizes the circuit corresponding to the function.

4. Applications

We consider two use cases involving garbled circuits: the secure two-party evaluation of common relational operations on private inputs, and the secure two-party evaluation of the SHA-256 hash function on an input that consists of two bit vectors (one from each of the two parties). This section focuses on the construction of high-level Python functions that can operate on bit vector inputs and how the definitions of these functions can be used with little or no modification to synthesize corresponding logical circuits. Once these circuits have been synthesized, they can be converted into an appropriate format and used within any MPC framework (such as JIGG[5]) that relies on garbled circuit protocols.

4.1 Relational Operations

Suppose that a programmer represents individual bits in Python using the integers `0` and `1`. One approach the programmer might use to implement an equality function on bits is by using Python's built-in arithmetic and logical operators on integer values, as illustrated in Figure 1. To synthesize a circuit that corresponds to this function, the programmer can simply import the circuitry library, introduce input and output type annotations, and add a decorator indicating that the function should be synthesized into a circuit. This is illustrated in Figure 2. As the interactive Python shell session in Figure 3 shows, the circuit synthesized from the function definition can be converted into the Bristol Fashion format using the bfcl library.

The programmer then has some options when using the function defined in Figure 1 to implement an equality function for *bit vectors*. In particular, they may adhere either to an imperative programming style (*e.g.*, by using language constructs such as loops) or to a functional programming style (*e.g.*, by using comprehensions and higher-order functions). Both of these are illustrated in Figure 4. One of these approaches may be preferable over the other in a particular scenario due to trade-offs such as performance, readability, or compatibility with optimizations. It might also be the case that the programmer chooses one based on their personal preference or their level of experience. The EDSL can accommodate

[1] The library is available at `https://pypi.org/project/circuit/`.

[2] The format definition and examples are available at `https://homes.esat.kuleuven.be/~nsmart/MPC/`.

[3] The library is available at `https://pypi.org/project/bfcl/`.

[4] The library is available at `https://pypi.org/project/circuitry/`.

[5] The library is available at `https://github.com/multiparty/jigg`.

```
def equal(x, y):
    return (x & y) | ((1 - x) & (1 - y))
```

Figure 1. Implementation of an equality function for a pair of bits.

```
from circuitry import bit, synthesize

@synthesize
def equal(x: bit, y: bit) -> bit:
    return (x & y) | ((1 - x) & (1 - y))
```

Figure 2. Implementation of an equality function for individual bits that is also synthesized into a circuit.

```
>>> from bfcl import circuit as bristol_fashion
>>> bristol_fashion(equal.circuit).emit()
5 7
2 1 1
1 1
2 1 0 1 2 AND
1 1 0 3 INV
1 1 1 4 INV
2 1 3 4 5 AND
2 1 2 5 6 LOR
```

Figure 3. Retrieval of the circuit synthesized in Figure 2 and its conversion into the Bristol Fashion format.

```
def equals(xs, ys):
    z = 1
    for i in range(len(xs)):
        z = z & equal(xs[i], ys[i])
    return z

def equals(xs, ys):
    from functools import reduce
    es = [equal(x, y) for (x, y) in zip(xs, ys)]
    return reduce((lambda e0, e1: e0 & e1), es)
```

Figure 4. Two implementations of an equality function for bit vectors; these exemplify the use of imperative (top) and functional (bottom) programming styles.

both of these approaches, and performing circuit synthesis using each of these definitions produces a circuit that can be evaluated to determine the equality of two bit vectors.

Note that the size of the input bit vector is not specified explicitly in either approach; this is determined automatically by the algorithm when it is executed (either for the purpose of determining a concrete output value or to synthesize a circuit). This ensures that even if the set of concrete circuit descriptions is arbitrarily large, the circuit that is used for a particular input can have a size that is tailored to that input. Similar high-level, dynamic-length implementations exist for all arithmetic operators. For example, the instance of `add32` used throughout the implementation in Figure 5 is an alias for a generalized `add_n` function that works for any two bit vectors of the same length.

4.2 SHA-256 Hash Function

By leveraging the EDSL, a working Python implementation of the SHA-256 algorithm implemented by the authors according to the FIPS 180-4 specification [4] can be used with almost no modification to perform circuit synthesis. Figure 5 presents the complete definition of the SHA-256 function that can both be evaluated on individual bit vectors and can be used to synthesize a corresponding logical circuit with the same input-output behavior.

The implementation in Figure 5 illustrates a number of features and benefits of the EDSL.

- Throughout the implementation, infix operators for common operations on bits and bit vectors are used. This includes logical operators (such as `&`, `|`, and `~`) and operators for shifting, rotating, and concatenating bit vectors (such as `>>` and `+`). These can be used because the EDSL overloads the methods for these operations within the definitions of the bit and bit vector classes.
- The number of iterations of the hash computation is based on how many 512-bit portions constitute the message as a whole. This means that the variant of the circuit that is built during synthesis depends on the input size specified at the time of synthesis. The input size can be specified by supplying to the `sha256` function a vector of input gates (rather than actual bits) of the desired length.
- Python lists and associated features (such as list comprehension syntax and slice notation) are used throughout to maintain tables of bits and to reuse bits as necessitated by the SHA-256 specification. When this algorithm is evaluated in a vector of input gates, these are effectively tables of references to circuit gates. This ensures that circuit gates are reused (at least when the information flow between table entries in the algorithm itself leads to such reuse).
- The `constant` class constructor from the circuitry library is used to ensure that the synthesized circuit includes gate inputs that correspond to the appropriate fixed bits from the initial hash and the table of constants. The *bitlist*[6] library is used to convert 32-bit integers into bit vectors of length 32.

5. Conclusions and Future Work

We have introduced a library that serves as an embedded domain-specific language for describing circuits. The library takes advantage of native host language features for code reuse and composition that include functions, data structures, comprehension syntax, and iteration constructs. We have also shown that circuits can be synthesized directly from function definitions by taking advantage of host language features such as decorators and type annotations.

The EDSL will be enhanced in the future in at least three ways. Hooks will be introduced into the class definitions that enable circuit optimizations to be applied as a circuit is being constructed (*e.g.*, the caching and reuse of references to internal gates or the application of algebraic laws). In addition to this, *abstract* interpretations of circuit construction operators will be implemented (*e.g.*, to allow the derivation of a polynomial that describes the size of a circuit rather than the circuit itself). Finally, a static analysis feature will be added that automatically determines whether a Python function definition can safely be converted to a circuit (*i.e.*, the body of the function does not have instances of branching constructs such as `if` in which the value of the condition expression can be influenced by an input value).

[6] The library is available at `https://pypi.org/project/bitlist/`.

```
from bitlist import bitlist
from circuitry import constants

def iteration(d_8_32s, m_64_8s):
    """
    Perform a single iteration of the hash computation over the current
    digest (consisting of 32 bit vectors with each having 8 bits) based on a
    message portion (consisting of 64 bit vectors with each having 8 bits)
    to produce an intermediate hash.
    """

    # Table of constants (64 individual bit vectors each having 32 bits).
    table = [constants(list(bitlist(i, 32))) for i in [
        1116352408, 1899447441, 3049323471, 3921009573,  961987163, 1508970993, 2453635748, 2870763221,
        3624381080,  310598401,  607225278, 1426881987, 1925078388, 2162078206, 2614888103, 3248222580,
        3835390401, 4022224774,  264347078,  604807628,  770255983, 1249150122, 1555081692, 1996064986,
        2554220882, 2821834349, 2952996808, 3210313671, 3336571891, 3584528711,  113926993,  338241895,
        666307205,   773529912, 1294757372, 1396182291, 1695183700, 1986661051, 2177026350, 2456956037,
        2730485921, 2820302411, 3259730800, 3345764771, 3516065817, 3600352804, 4094571909,  275423344,
        430227734,   506948616,  659060556,  883997877,  958139571, 1322822218, 1537002063, 1747873779,
        1955562222, 2024104815, 2227730452, 2361852424, 2428436474, 2756734187, 3204031479, 3329325298
    ]]

    # Functions used during the hash computation.
    Ch = lambda x, y, z: (x & y) ^ ((~x) & z)
    Maj = lambda x, y, z: ((x & y) ^ (x & z)) ^ (y & z)
    Sigma_0 = lambda bs: ((bs >> {2}) ^ (bs >> {13})) ^ (bs >> {22})
    Sigma_1 = lambda bs: ((bs >> {6}) ^ (bs >> {11})) ^ (bs >> {25})
    sigma_0 = lambda bs: ((bs >> {7}) ^ (bs >> {18})) ^ (bs >> 3)
    sigma_1 = lambda bs: ((bs >> {17}) ^ (bs >> {19})) ^ (bs >> 10)

    w = [] # Message schedule.
    for j in range(16):
        w.append(m_64_8s[j*4] + m_64_8s[j*4+1] + m_64_8s[j*4+2] + m_64_8s[j*4+3])
    for j in range(16, 64):
        w.append(add32(add32(add32(sigma_1(w[j-2]), w[j-7]), sigma_0(w[j-15])), w[j-16]))

    wv = [i32 for i32 in d_8_32s] # Eight 32-bit working variables.
    for j in range(64):
        c = add32(add32(Ch(wv[4], wv[5], wv[6]), table[j]), w[j])
        t1 = add32(add32(wv[7], Sigma_1(wv[4])), c)
        t2 = add32(Sigma_0(wv[0]), Maj(wv[0], wv[1], wv[2]))
        wv = [add32(t1, t2), wv[0], wv[1], wv[2], add32(wv[3], t1), wv[4], wv[5], wv[6]]

    return [add32(d_8_32s[j], wv[j]) for j in range(8)] # Return intermediate hash.

def sha256(message):
    """
    Accept a list ‘message‘ of bit vectors each having 8 bits (for a total number that is a multiple of 512),
    and compute a SHA-256 message digest consisting of 32 individual bit vectors each having 8 bits.
    """

    # Initial hash value represented as eight individual bit vectors each having 32 bits.
    digest = [constants(list(bitlist(i, 32))) for i in [
        1779033703, 3144134277, 1013904242, 2773480762,
        1359893119, 2600822924,  528734635, 1541459225
    ]]

    # Perform hash computation for appropriate number of iterations depending on the message length.
    for i in range(len(message) // 64):
        digest = iteration(digest, message[(i * 64) : (i * 64) + 64])

    # Turn eight individual bit vectors each having 32 bits into 32 individual bit vectors each having 8 bits
    # using the ‘/‘ operator that has been overloaded to act as a splitting operation on bit vectors.
    return bits([b for bs in digest for b in bs]) / {8}
```

Figure 5. Python implementation of the SHA-256 hash function (for properly padded inputs of a size divisible by 512) that conforms to the FIPS 180-4 specification [4] and that supports both evaluation on inputs and (without modification) circuit synthesis via the circuitry library.

References

[1] K. D. Albab, R. Issa, A. Lapets, P. Flockhart, L. Qin, and I. Globus-Harris. Tutorial: Deploying Secure Multi-Party Computation on the Web Using JIFF. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 3–3, McLean, VA, USA, September 2019. URL `https://doi.org/10.1109/SecDev.2019.00013`.

[2] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, page 1216–1225, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450311991. URL `https://doi.org/10.1145/2228360.2228584`.

[3] A. Bestavros, A. Lapets, and M. Varia. User-Centric Distributed Solutions for Privacy-Preserving Analytics. *Communications of the ACM*, 60(2):37–39, February 2017. URL `https://doi.org/10.1145/3029603`.

[4] Q. Dang. Changes in Federal Information Processing Standard (FIPS) 180-4, Secure Hash Standard. *Cryptologia*, 37(1):69–73, 2013. URL `https://doi.org/10.1080/01611194.2012.687431`.

[5] M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison-Wesley, Upper Saddle River, NJ, 2011. ISBN 978-0321712943.

[6] S. Garhwal, M. Ghorani, and A. Ahmad. Quantum Programming Language: A Systematic Review of Research Topic and Top Cited Languages. *Archives of Computational Methods in Engineering*, Dec. 2019. ISSN 1886-1784. URL `https://doi.org/10.1007/s11831-019-09372-6`. ZSCC: 0000000.

[7] O. Goldreich. Cryptography and cryptographic protocols. *Distributed Comput.*, 16(2-3):177–199, 2003. URL `https://doi.org/10.1007/s00446-002-0077-1`.

[8] S. Golson and L. Clark. Language Wars in the 21st Century: Verilog versus VHDL—Revisited. In *Synopsys Users Group (SNUG)*, 2016.

[9] M. Hastings, B. Hemenway, D. Noble, and S. Zdancewic. SoK: General Purpose Compilers for Secure Multi-Party Computation. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1220–1237, 2019. URL `https://doi.org/10.1109/SP.2019.00028`.

[10] A. Lapets and M. Rötteler. Abstract Resource Cost Derivation for Logical Quantum Circuit Descriptions. In *Proceedings of FPCDSL 2013: The 1st Workshop on Functional Programming Concepts in DSLs*, Boston, MA, USA, September 2013. URL `https://doi.org/10.1145/2505351.2505358`.

[11] A. Lapets, M. Silva, M. Thome, A. Adler, J. Beal, and M. Rötteler. QuaFL: A Typed DSL for Quantum Programming. In *Proceedings of FPCDSL 2013: The 1st Workshop on Functional Programming Concepts in DSLs*, Boston, MA, USA, September 2013. URL `https://doi.org/10.1145/2505351.2505357`.

[12] A. Lapets, F. Jansen, K. D. Albab, R. Issa, L. Qin, M. Varia, and A. Bestavros. Accessible Privacy-Preserving Web-Based Data Analysis for Assessing and Addressing Economic Inequalities. In *Proceedings of ACM COMPASS 2018: First Conference on Computing and Sustainable Societies*, San Jose, CA, USA, June 2018. URL `https://doi.org/10.1145/3209811.3212701`.

[13] 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*, pages 359–376, 2015. URL `https://doi.org/10.1109/SP.2015.29`.

[14] L. Qin, P. Flockhart, A. Lapets, K. D. Albab, M. Varia, S. Roberts, and I. Globus-Harris. From Usability to Secure Computing and Back Again. In *Proceedings of the 15th Symposium on Usable Privacy and Security (SOUPS)*, Santa Clara, CA, USA, August 2019. URL `https://dl.acm.org/doi/10.5555/3361476.3361490`.

[15] A. Shamir. How to share a secret. *Communications of the ACM*, 22 (11):612–613, 1979. URL `https://doi.org/10.1145/359168.359176`.

[16] L. Truong and P. Hanrahan. A Golden Age of Hardware Description Languages: Applying Programming Language Techniques to Improve Design Productivity. In B. S. Lerner, R. Bodík, and S. Krishnamurthi, editors, *3rd Summit on Advances in Programming Languages, SNAPL 2019, May 16-17, 2019, Providence, RI, USA*, volume 136 of *LIPIcs*, pages 7:1–7:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019. URL `https://doi.org/10.4230/LIPIcs.SNAPL.2019.7`.

[17] X. Wang, A. J. Malozemoff, and J. Katz. EMP-toolkit: Efficient MultiParty computation toolkit. `https://github.com/emp-toolkit`, 2016.

[18] A. C. Yao. Protocols for Secure Computations. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 160–164, Washington, DC, USA, 1982. IEEE Computer Society. URL `http://dx.doi.org/10.1109/SFCS.1982.88`.

[19] A. C. Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167, 1986. URL `https://doi.org/10.1109/SFCS.1986.25`.

[20] S. Zahur and D. Evans. Obliv-C: A Language for Extensible Data-Oblivious Computation. Cryptology ePrint Archive, Report 2015/1153, 2015. `https://eprint.iacr.org/2015/1153`.