

Automated Quantum Circuit Generation for Computing Inverse Hash Functions

Elena R. Henderson^a, Jessie M. Henderson^a, William V. Oxford^b, and Mitchell A. Thornton^a

^aDarwin Deason Institute for Cyber Security, Southern Methodist University,
6425 Boaz Lane Dallas, TX 75205, USA

^bAnametric, Inc., Austin, TX 78652, USA

ABSTRACT

Several cryptographic systems depend upon the computational difficulty of reversing cryptographic hash functions. Robust hash functions transform inputs to outputs in such a way that the inputs cannot be later retrieved in a reasonable amount of time even if the outputs and the function that created them are known. Consequently, hash functions can be cryptographically secure, and they are employed in encryption, authentication, and other security methods. It has been suggested that such cryptographically-secure hash functions will play a critical role in the era of post-quantum cryptography (PQC), as they do in conventional systems. In this work, we introduce a procedure that leverages the principle of reversibility to generate circuits that invert hash functions. We provide a proof-of-concept implementation and describe methods that allow for scaling the hash function inversion approach. Specifically, we implement one manifestation of the algorithm as part of a more general automated quantum circuit synthesis, compilation, and optimization toolkit. We illustrate production of reversible circuits for crypto-hash functions that inherently provide the inverse of the function, and we describe data structures that increase the scalability of the hash function inversion approach.

Keywords: automated quantum circuit synthesis, cryptographic hash function, cryptography, inverse hash function, hash reversal

1. INTRODUCTION

Hash functions transform inputs of one form to outputs of another via a mathematical function.¹ Ideally, every unique input is transformed to a unique output, and robust hash functions minimize collisions, which occur when two differing inputs map to the same output. Since the 1970s, hundreds of hash functions have been used in applications such as digital signatures, password tables, key derivation, and blockchains.^{2–6} Hash functions designed for such cryptographic applications must satisfy specific properties; for example, many hash functions are designed to be *one-way* such that it is straightforward to move from input I to output O , but very difficult to recover I when only O and the hash function are known. The process of moving from output to input is termed *computation of an inverse hash function* or *hash reversal*, and efficient techniques for reversing cryptographically-sound hash functions are widely researched.

We have developed a procedure for synthesizing reversible circuits that constitute hash function inverses. The remainder of this article describes this method as follows: We begin with background on cryptographic hashes, logical reversibility, and the toolkit that we use to automate production of inverse hash function circuits. We then describe the procedure for generating the inverse hash circuits, and explain how they work. Next, we present experimental results; as all of the functions we consider are small by cryptographic standards, we close by discussing avenues for further work, including additional applications and scalable function representations.

Send correspondence to ERH: erhenderson@smu.edu. Further author information: JMH: hendersonj@smu.edu, WVO: oxford@anametric.com, MAT: mitch@smu.edu

2. BACKGROUND

2.1 Cryptographic Hash Functions

In this article, we limit our scope to hash functions defined for bitstrings. Specifically, we consider hash functions that convert a fixed-length bitstring input to a fixed-length bitstring output. There are several criteria that are useful for general-purpose hash functions; for example, it is generally desirable for differing inputs to map to differing outputs in as many cases as possible.¹ Requirements are more stringent for *cryptographic hash functions*, which are those for which input bitstrings should remain unobtainable even when the corresponding output bitstrings and the function that generated them are known.

The study of cryptographic hash functions is both rich and complex, so a comprehensive survey is beyond the scope of this article.^{2,3,7} Here, it suffices to note that there are several properties that are oft-considered requirements for cryptographic hash functions,^{2,3,7} including the following four. First, a hash function is defined as *one-way* if it is difficult to determine an input from an output when both the output and the function in question are known.³ Second, a hash function is defined as *second-preimage resistant* if—when given one input—it is difficult to find a second input that maps to the same output.³ Third, a hash function is defined as *collision resistant* if it is difficult to find two inputs that have the same output.³ Fourth and finally, a hash function is defined as satisfying the *Avalanche Criteria* if it meets a two-part specification.^{3,8} First, each output should differ from its corresponding input with a Hamming distance that is at least half of the number of output bits. And second, for each pair of inputs that differs by a Hamming distance of one, the corresponding outputs should differ by a Hamming distance of at least half of the number of output bits.

In addition to properties specifying the relationship between inputs and outputs, cryptographic hash functions are generally understood to require hundreds of bits in inputs and outputs, because such bitstring lengths make brute force attacks impractical.² But in this article, we will consider much smaller hash functions of at most 8-bits for two reasons. First, we aim to present conceptual examples, which quickly become prohibitive as the number of bits in a hash function increases. Second, the function specification format that we use for our experimental results (`.pla` format; see Section 3.1) accommodates some functions with hundreds of inputs and outputs, but does not scale for functions with the properties of cryptographic hash functions. Consequently, in Section 5, we discuss alternative function specification formats that would allow for representing large cryptographic hash functions.

2.2 Logical Reversibility

Logically-reversible circuits are those for which each output has a unique input: knowing solely the outputs and the function that was applied to generate them allows for obtaining the inputs that led to those given outputs. Unsurprisingly, logically-reversible circuits must be comprised of logically-reversible operations, which are perhaps best explained with an example.

Consider a two-input AND operation, which has the truth table and circuit diagram specified in Figure 1. Given the output $r = 0$ and the information that an AND operation was applied, one cannot know what the inputs were, since there are three possible input combinations associated with $r = 0$: 1) $p = 0$ and $q = 0$; 2) $p = 0$ and $q = 1$; and 3) $p = 1$ and $q = 0$. Consequently, the AND operation is not reversible. Contrast this with the three-input Toffoli operation in Figure 2 and the outputs $s = 1$, $t = 1$, and $u = 0$. The only possible set of inputs that produces such outputs is $p = 1$, $q = 1$, and $r = 1$. Thus, the Toffoli operation is reversible, and we see that adding inputs and outputs can make an irreversible function reversible.⁹

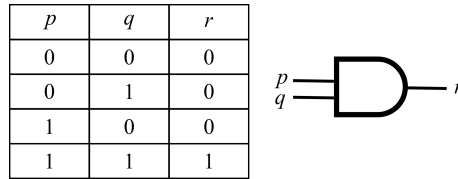


Figure 1. Truth table and circuit representation for an irreversible AND operation.

p	q	r	s	t	u
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

Figure 2. Truth table and circuit representation for a reversible Toffoli operation. The Toffoli operation is a controlled operation, in which a NOT is applied to r if and only if both p and q are one. p and q are called the controls, and r is termed the target of the operation. The values of controls do not change (meaning $s = p$ and $t = q$), but both values are included as outputs such that the operation is reversible.

2.3 Brief Introduction to *MustangQ*

Our inverse hash circuit-generation procedure is one application of a broader software design effort, so we briefly introduce *MustangQ* before describing the reversal process. *MustangQ* is a synthesis, compilation, and optimization toolkit that transforms functions from a representation suitable for synthesizing classically-specified circuits to circuit descriptions appropriate for quantum computers.^{10–15} There is often more than one way to represent a classically-specified function as a quantum circuit, and previous work has shown that different strategies lend themselves to different circuit properties, including number of required qubits, number of required gates, and required gate types.^{10,11,13} These features affect the viability of running circuits on contemporary quantum hardware, which currently has relatively few qubits, all of which suffer from significant amounts of noise.^{16–18} Therefore, understanding ideal use cases and optimizations for varying quantum circuit types is important, and *MustangQ* automates the rote circuit generation tasks required to efficiently complete such analysis.

Our hash reversal procedure depends upon only a subset of *MustangQ*’s functionality, and for the remainder of this paper, we will discuss only that subset. These capabilities are motivated by the fact that quantum and classical computers, while both involving circuits, have a number of important differences. One such difference is that some quantum circuit synthesis methods require reversible function specifications.¹⁹ As a result, *MustangQ* has the ability to transform irreversible functions into reversible forms. So, when hash functions are provided as inputs, *MustangQ* can synthesize reversible function representations that are then embodied in synthesized circuits.

The three gates used in this article for inverse hash circuits are NOT or Pauli-**X** gates (illustrated in Figure 3), Controlled-Not or C-NOT gates (illustrated in Figure 4), and Toffoli gates (illustrated in Figures 2 and 5). Toffoli gates are either ‘true’ Toffolis with only two controls (Figure 2), or generalized Toffolis with three or more controls (Figure 5). It is worth noting that this specific gateset is applicable for both quantum and classical circuits. Thus, while *MustangQ*’s primary use is generation of quantum circuits—meaning this article will use the notation and terminology standard for that field—it is worth noting that the circuits described herein can also be understood classically. Future work (briefly addressed in Section 5) ought to investigate how these inverse hash circuits might be used in a quantum-specific way to parallelize the output-to-input procedure.

p	q
0	1
1	0

Figure 3. Truth table and circuit representation for a reversible NOT operation. The first representation is common in classical circuits, while the second and third are used in quantum circuits.

p	q	r	s
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	1

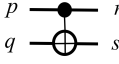


Figure 4. Truth table and circuit representation for a reversible Controlled-NOT operation.

p	q	r	s	t	u	v	w
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	0
0	0	1	1	0	0	1	1
0	1	1	0	0	1	1	0
0	1	1	1	0	1	1	1
1	0	0	0	1	0	0	0
1	0	0	1	1	0	0	1
1	0	1	0	1	0	1	0
1	0	1	1	1	0	1	1
1	1	0	0	1	1	0	0
1	1	0	1	1	1	0	1
1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	0

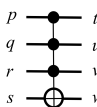


Figure 5. Truth table and circuit representation for a generalized Toffoli with three controls: p , q , and r are controls for target s ; t , u , and v maintain the values of p , q , and r ; and $w = \sim s$ when p , q , and r are all one.

3. COMPUTING INVERSE HASH FUNCTIONS

To obtain a circuit specification that constitutes an inverse hash function, there are three broad steps. First, prepare the given hash function as an electronic design file. Second, synthesize a circuit from that function using the ESOP Synthesis Method, which generates reversible circuits even from irreversible function specifications. Third, reverse the synthesized circuit by simply flipping the sequential order of the gates comprising the circuit.

3.1 Preparing the Hash Function

The first step is to specify the hash function as an electronic design file. There are several acceptable formats, including structural Verilog HDL netlists or decision diagrams, such as Binary Decision Diagrams (BDD). In Section 5, we briefly discuss such formats, as they present scalable approaches to bitstring function representation. But in most of this paper, we focus on the `.pla` format, which was originally devised as a compact textual representation for specifying digital logic programmable logic arrays for physical implementation in electronic circuits and, in particular, as a file format that allowed specifications to be minimized.^{20,21} We selected `.pla` as the primary format in this article for two reasons. First, `.pla` files are relatively straightforward and standardized, providing a user-friendly—and parser-friendly—way to specify bitstring functions. Second, we have a sizeable set of benchmark function specifications in either `.pla` format or in other tabular forms that are readily converted to `.pla` files, including collections maintained via RevLib.²² So, the first step in computing an inverse hash function is to prepare that function as a `.pla` file.

The `.pla` format can describe arbitrary functions of the form $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$, where $\mathbb{B} = \{0, 1\}$, and n and m are integers. Each row of a `.pla` file consists of $n + m$ bits: the first n bits are one input, and the remaining m bits constitute a corresponding output. Additionally, each row can represent multiple input/output pairs, using dashes ('-'). Specifically, a dash in an input bitstring indicates that either a 0 or a 1 is an input that corresponds to the associated output. And a dash in an output bitstring indicates that either 0 or 1 may be used interchangeably in that particular output. Thus, functions that have cube-covering overlap are appropriate for efficient representation in `.pla` format.

Figure 6 illustrates a trivial function represented as a `.pla` file. The full tabular representation in Subfigure B is simplified using dashes to the form in Subfigure C, and while this trivial example has minimal simplification, `.pla` files that represent larger functions with significant amounts of cube-covering overlap can be significantly reduced in size. Consequently, using `.pla` files is advantageous because they are intuitive and can efficiently represent even very large functions, so long as those functions have sufficient cube-covering overlap; for example, `.pla` files can efficiently represent functions with over one hundred inputs and outputs.*

A)

Input 1	Input 2	Output
0	0	0
0	1	0
1	0	0
1	1	1

B)

```
# Full .pla
.i 2 # Number of inputs
.o 1 # Number of outputs
.p 4 # Number of lines in file
00 0
01 0
10 0
11 1
.e # End of file
```

C)

```
# Simplified .pla
.i 2 # Number of inputs
.o 1 # Number of outputs
.p 3 # Number of lines in file
0- 0
-0 0
11 1
.e # End of file
```

Figure 6. Subfigure A illustrates a simple function translated to `.pla` format in Subfigure B. Subfigure C illustrates a simplification of the `.pla` with the introduction of hyphens (-). The hyphen in line 1 indicates that, regardless of the value of the second input, when the first input is zero, the output is also a zero. Similarly, the hyphen in line 2 indicates that, whenever the second input is a zero, the output is also zero. For this trivial function, adding such hyphens removes only one line from the `.pla`, but for larger functions with more overlap, the reduction of the number of lines can be significant.

3.2 Synthesizing Hash-Function-Representing Circuits

After preparing an appropriate function specification, the Exclusive-Or-Sum-of-Products (ESOP) Synthesis Method is used to produce a reversible circuit.¹⁹ The Method—as with the rest of the *MustangQ* toolkit—was originally developed with quantum circuits in mind, and produces a circuit with the original number of inputs plus the original number of outputs. The first step is generally to produce a minimized Exclusive-Or-Sum-of-Products, or ESOP, from the `.pla` representation of a hash function. An ESOP is a function with inputs and outputs specified as bitstrings whose individual bits are interpreted as being combined via logical conjunctive AND operators. These bitstrings (termed products) are then summed with disjunctive exclusive-OR operators.¹⁹ Evaluating the resulting expression with a given input bitstring produces that input’s corresponding output bitstring. Although most functions will not natively be in ESOP form, they can be converted via an automated process, whereupon the rows in a tabular representation of bitstring functions are the products of the ESOP.²³ This process is termed ESOP minimization, and we use EXORCISM-4 to implement it.

The ESOP Synthesis Method produces a quantum circuit by mapping each product to a Toffoli gate in two steps. First, qubits are added to the circuit: one qubit is added for each input variable, and one qubit is added for each output variable. Qubits must be added for both the input and output variables, because circuits generated with ESOP synthesis do not overwrite the input values with output values. Second, for each product in the table, and for each output variable value of 1 therein, a Toffoli gate is added to the circuit. The controls of the Toffoli gate are determined by the inputs of the current product: an input variable value of 1 means a control is placed on the corresponding input qubit, while an input value of 0 means a negative control is placed on the corresponding input qubit. A negative control is simply a control with a NOT gate on either side; the NOT gates reverse the qubit’s polarity before the control and restore it after the Toffoli gate has executed. The Toffoli gate’s target is placed on the output qubit corresponding to the output variable whose value is 1.

*For example, the file `frg2` from RevLib contains 143 inputs and 139 outputs.²²

Before turning to an example and experimental results, we include two clarifying points. First, we explain how this process generates reversible circuits. By using only reversible operations (recall from Section 2.2 that NOT gates, C-NOT gates, and Toffoli gates are reversible) and a qubit for each input *and* each output, the ESOP Synthesis Method preserves the input values throughout the computation. This interpretation of the function is consequently reversible, because each set of input values is effectively appended—unchanged—to each set of output values at the end of circuit computation. Second, we note that the core approach of the ESOP Synthesis Method also works for tabular functions that have not been minimized to ESOP form. Specifically, the process of adding gates as specified above works to produce reversible circuits from functions in non-minimized `.pla` form, but at the cost of far more gates. Consequently, it is generally advantageous to use the ESOP Synthesis Method as intended, with functions in ESOP form; however, minimization is a computationally-intensive process, so it is worth noting that the step could be skipped for large hash functions at the cost of substantially less efficient circuits.

3.3 Computing the Inverse Hash Function Circuits

Finally, the circuit specification generated in the above step can simply be reversed, meaning that the gate order is swapped. For example, if gates are numbered $1, 2, 3, \dots, n-1, n$, the reversed circuit would have gate order $n, n-1, \dots, 3, 2, 1$. This reversed circuit then embodies the inverse hash function, as illustrated in the next section.

4. EXAMPLE AND EXPERIMENTAL RESULTS

We begin this section with an example of synthesizing a circuit for a trivial four-bit hash function. Subfigure A of Figure 7 presents the hash function in tabular form; it is an extremely weak hash function, both because of its size and because it does not satisfy the Avalanche Criteria. But for purposes of considering an illustrative example, we did not need to craft a strong hash function because—besides requiring a longer completion time and larger circuits—the process does not change with hash function strength. Applying *MustangQ*’s ESOP Synthesis Method produces the circuit of Subfigure B of Figure 7. Figure 8 again illustrates the circuit, this time with the input and output values labelled with a particular input/output pair: 0110 and 1001. As noted in Section 3.2, the input values pass through the circuit unchanged, while the outputs evolve from a state of all zeros to the values associated with the given input. Reversing the order of the gates in the circuit specification allows for working backwards from the output values to obtain the input values. This input-retrieval process is illustrated in Figure 9, to which we now turn.

We begin on the left-hand side, with variables w, x, y , and z representing the unknown input values. The numerical values on the left-hand side represent the known output values corresponding to the input values we seek. On the right-hand side, variables w, x, y , and z again represent those unknown input values, while the initial output values are set to zero.

It is worth emphasizing two points. First, the values of w, x, y , and z are the same on both the left and right sides of the circuit, because the paired NOT gates restore input values after changing them.[†] While this input immutability is not a requirement for the reversal process, it is a feature of the *MustangQ* toolkit that is useful in synthesizing circuits more generally, so we leverage it here. Second, the initialization of output values to zero is not a requirement for function reversal, but is again a *MustangQ* feature that is also convenient here. It would be possible to use this same reversal procedure with output values initialized to non-zero values; what is important is that the initialization values be a known part of the synthesis process. The input values can be determined by working backwards; because the circuits produced from reversible functions are reversible, we can ascertain what must have happened at each step to obtain the known outputs. Figure 9 denotes the numbers that are relevant to obtain each of the four input values; for the sake of brevity, we describe the process for only two of the four.

Consider the red gate identified with a red a . Line VIII has a final output value of 1, but it started as a 0. By definition of the C-NOT operation, this means that red gate a flipped the value of line VIII, such that the

[†]The NOT gates that occur to the left of the C-NOT gates in Figure 9 are the input restoration gates that are not strictly necessary; in Figure 9, these gates are lavender-colored.

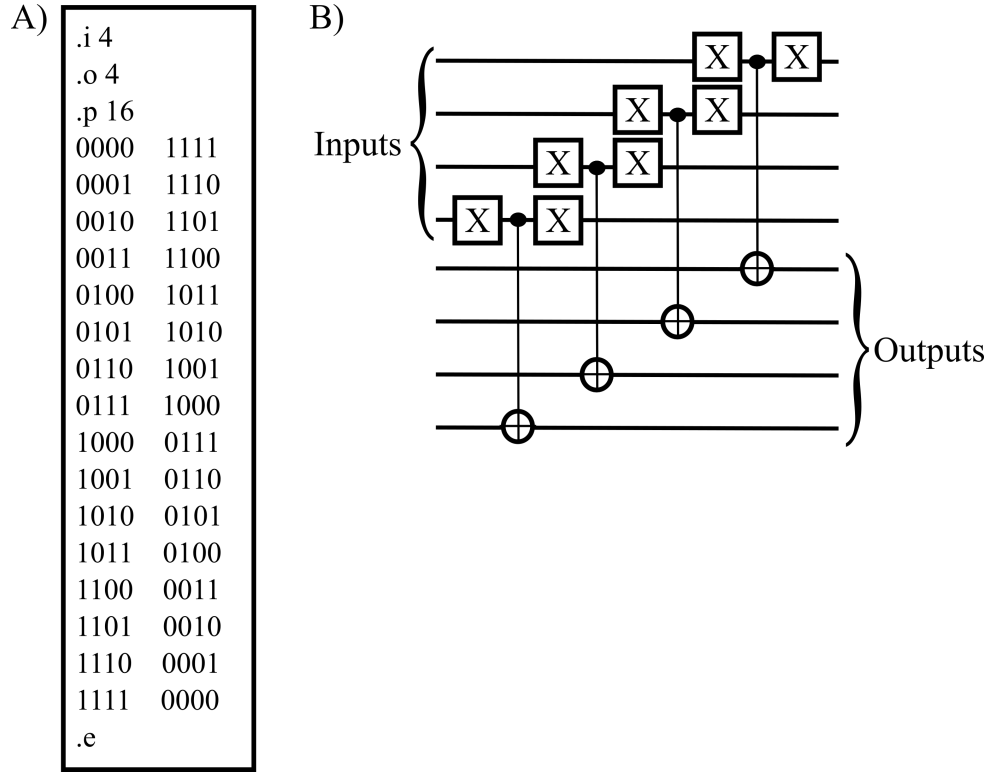


Figure 7. Subfigure A illustrates a four-bit hash function, while Subfigure B illustrates the circuit synthesized from this function using the ESOP Synthesis Method of *MustangQ*.

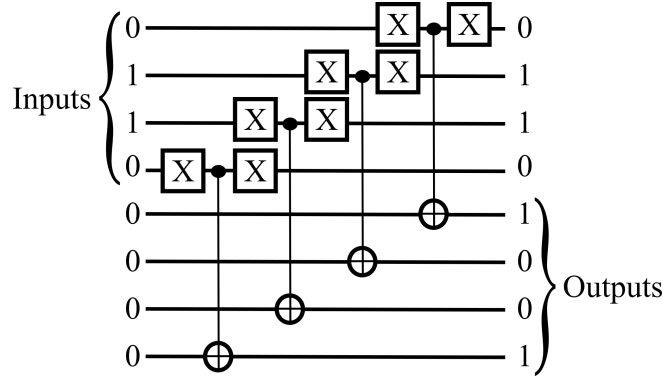


Figure 8. Generating the output 1001 from input 0110. Notice that the inputs remain unchanged, while the outputs contain the desired calculation.

control value must have been a 1, denoted by the dark blue 1 to the right of the control on line IV. Tracing line IV to the right, we find a NOT gate. Since the result of that gate must be a 1, we know that the input to the NOT was 0, and consequently, we find that input z had a starting value of 0.

As our second example, consider the light blue gate identified as b . Blue gate b has a target on line VII which has both a final and starting value of 0. This means that the control for blue gate b (located on line III), must have had an input of 0, denoted by the pink zero to the right of the control. Tracing from that control to the beginning of line III leads to a NOT gate; since the result of the NOT is 0, the input must have been 1, meaning that $y = 1$.

Such manual validation is possible only for very small functions. For even moderately larger functions (such

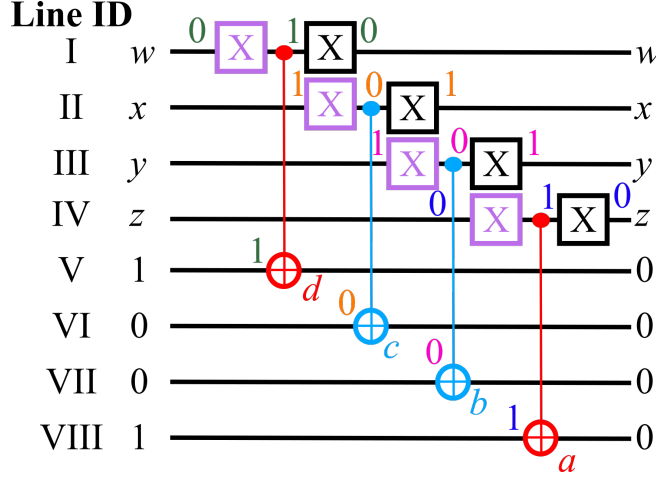


Figure 9. Working backwards from output values 1001 to obtain the input values 0110. The numbers closest to the right-hand side on lines I, II, III, and IV are the deduced input values: $w = 0$, $x = 1$, $y = 1$, and $z = 0$. The numbers indicating intermediate values of each line are color-coded such that values related to w ’s reversal are in pine green, values related to x ’s reversal are in orange, values related to y ’s reversal are in pink, and values related to z ’s reversal are in dark blue. Gates in red denote that the gate flipped its output value, while gates in light blue denote that the output was not changed. The accompanying red and light blue letters serve to identify particular gates in the text description.

as those in Table 1, validation was accomplished by combining the circuit specification in the ‘forward’ (*i.e.*, input to output) direction with that for the ‘reversed’ (*i.e.*, output to input) direction. We used Verilog HDL simulation to verify that all of the values entered on the left side matched those on the right side; when the combined forward and reversed circuits act as identity for all possible input/output pairs, reversal is proven successful. We have inverted the hashes presented in Table 1 and have validated the results for each.

The entries in Table 1 are all very small hash functions that are intended as preliminary, proof-of-concept results; they are sufficiently small that we were able to use the .pla format, to validate exhaustively, and even to manipulate a few of the circuits by hand during development. These results illustrate that our approach is worthy of further exploration for four reasons.

First, note that none of these times are prohibitive, nor even close: the average ESOP minimization time is about 0.46 seconds, the average circuit synthesis time after minimization is about 0.089 seconds, and the average circuit synthesis time without minimization is about 0.256 seconds. All of the known temporal results in the table require under ten minutes of CPU time, and this is with a preliminary implementation that could undoubtedly be optimized, including via more efficient data structures. Such optimization is important, as it is worth noting that the wall clock times for processing these circuits are often quite a bit longer than the required CPU time; for example, the 8-bit hash that was designed to satisfy the Avalanche Criteria, the wall clock time is on the order of minutes. This is a consequence of time-consuming file I/O operations that are in part due to the human-readable format of .pla files, as well as the frequent write operations used to validate the circuit synthesis and report on its statistics. Consequently, we are motivated to explore alternative formats for representing and processing the functions, as described in Section 5.

Second, it is worth noting that although these hash functions are small, *MustangQ* has been successfully tested at scale in situations where the functions are not hash functions,^{10,11,13} and we have yet to test a function for which *MustangQ* cannot generate a circuit. While this is certainly not to say that such functions do not exist, it is worth emphasizing that *MustangQ*’s capabilities are proven far beyond functions with 8 inputs and outputs.^{10,11,13} Additionally, even with hash functions, we have tested *MustangQ* with slightly larger ‘hashes’ of up to 21 bits. So, although we opted not to report those results here (the hashes are trivial, with outputs that are simply offsets of the inputs), we emphasize that the experiments of Table 1 do not express the extent of *MustangQ*’s capabilities. What does remain to be shown is that our method for inverting hashes will scale as larger hash functions—and not just larger functions in general—are synthesized via *MustangQ*.

Third and related to minimization, it is noteworthy that adding bits—and complexity of the function—causes a steep increase in the amount of time required for ESOP minimization. Although the total number of inputs and outputs increases by a factor of at most 2, the corresponding increase in ESOP minimization time is a factor of at least 172. This increase seems more substantial when contrasted with the time increase required for the circuit synthesis process itself. Specifically, for construction of the circuit after ESOP minimization, there is a factor of about 58 between the largest and smallest times, and for circuit synthesis without ESOP minimization, there is a factor of about 61 between the same. As both of these are an order of magnitude below the corresponding increase for ESOP minimization time, optimizing or eliminating the ESOP minimization may be necessary when scaling the inverse hash computation procedure.

Fourth and finally, we note that although ESOP minimization does impose a temporal cost, it offers meaningful gains in the number of gates required. Synthesizing without first minimizing always imposed a gate count penalty, and the average number of gates per circuit increased from about 273 with minimization to about 514 without.

5. CONCLUSION AND FUTURE WORK

Efficiently reversing hashes has a number of applications, and while some lend themselves to more pernicious consequences than others, all of them are important to the cybersecurity community. Consequently, this article is an important step that also raises several avenues of further research, which are primarily focused on the goal of testing the procedure at scale. We close by briefly considering two: further exploration of applications—including how the relevant hashes might be represented in a tabular format for *MustangQ*—and further exploration of scalable function representation methods.

5.1 Exploring Applications

As is widely recognized in works such as Refs. 1–7 and their hundreds of references, hash functions have a host of applications, from participating in blockchain ecosystems (including cryptocurrency mining) to decrypting website credentials to impersonating cryptographic signatures. *MustangQ*’s circuit generation directly calculates an inverse hash, thus bypassing the massive set of forward-hash computations involved in a brute-force search for any of these applications. But leveraging the toolkit’s full value may require situation-dependence. For example, the *MustangQ* approach is especially suitable for tasks involving fixed-size hashes—such as those used in blockchains—because the hash reversal circuit would not need to be resynthesized in real time.

To illustrate this, we briefly describe how *MustangQ* might apply to a blockchain application, thus motivating the need for further application-specific research. Consider cryptocurrency mining, which involves attempting to reverse a hash function that is calculated on a specific blockchain database. Once the hash function has been reversed, it is used to create a “Proof-of-Work” (PoW) that validates the integrity of the given blockchain entry. This mining process typically requires significant computational resources, given that classical computers are often required to perform a brute-force search—which requires computing all of the possible forward hash results for a given blockchain entry—to obtain the reverse of the blockchain signature. As mentioned above, applying our method would avoid such an exhaustive search. Furthermore, if the inverse hash circuit were utilized as a quantum—and not a classical—circuit, it might be possible to obtain information about multiple sets of inputs simultaneously, and the feasibility of this on an application-specific basis could prove fruitful.

5.2 Exploring Alternative Function Representations

There are several reasons that alternative function representations are important for our procedure, including the spatial feasibility of representing the functions, the temporal feasibility of parsing and pre-processing the functions, and the temporal feasibility of then applying the actual circuit synthesis algorithm. We have not yet encountered issues with the algorithmic approaches of *MustangQ*,^{10–15} so here we will focus on the first two issues.

First, space constraints: there are many functions that are too large to be efficiently processed as .pla files. Despite the size reduction available to functions with cube-covering overlap, .pla files are inherently limited, because the size of .pla specifications for functions without substantial cube-covering overlap grows

Table 1. Statistics for hash function circuits with five notes: 1) The 8-bit hash was custom designed to satisfy the Avalanche Criteria. 2) The time to synthesis was recorded as CPU time when running on a HP DL380 server with a 16 Core Intel Xeon processor with 2.9GHz and 380GB of RAM. 3) The first set of times and gate counts is from runs using ESOP minimization. As such minimization could be removed if scalability were an issue, the second set of times and gate counts are for synthesis without ESOP minimization. 4) All synthesis results include a standard optimization that removes superfluous NOT gates. 5) EXORCISM-4 reports timing results in intervals of 0.01 seconds. So, “< 0.01 s” indicates when the minimization was faster than that time, and the “*” indicates that EXORCISM-4 ran successfully, but reported its time-counter had overflowed.

Function Name	Number of Inputs	Number of Outputs	ESOP Minimization Time	Circuit Synthesis Time (With Minimization)	Circuit Gate Count (With Minimization)	Circuit Synthesis Time (No Minimization)	Circuit Gate Count (No Minimization)
4-bit AES S-Box	4	4	< 0.01 s	0.005888 s	29	0.015233 s	60
PRESENT S-Box	4	4	< 0.01 s	0.006541 s	29	0.015096 s	60
DES S-Box 1	6	4	0.01 s	0.025588 s	105	0.071742 s	246
DES S-Box 2	6	4	0.01 s	0.019397 s	86	0.072220 s	246
DES S-Box 3	6	4	< 0.01 s	0.017789 s	82	0.071665 s	246
DES S-Box 4	6	4	0.01 s	0.025794 s	101	0.072256 s	249
DES S-Box 5	6	4	0.01 s	0.025501 s	113	0.071608 s	246
DES S-Box 6	6	4	0.01 s	0.023222 s	90	0.071501 s	246
DES S-Box 7	6	4	< 0.01 s	0.023332 s	90	0.071921 s	246
DES S-Box 8	6	4	0.01 s	0.016621 s	103	0.071770 s	248
AES S-Box	8	8	1.72 s	0.326583 s	935	0.916981 s	1532
AES Inverse S-Box	8	8	1.29 s	0.301835 s	869	0.882464 s	1532
8-bit hash	8	8	1.06 s	0.344246 s	918	0.925617 s	1532

exponentially with respect to the number of inputs and outputs. As hash functions may not have substantial overlap in input/output pairs, only hashes with small numbers of inputs and outputs may be represented as .pla files.[‡]

Therefore, another avenue for future work is to explore efficient production of hash functions represented as

[‡]For example, a hash function with an input/output size of 256 bits could require 1.16e77 lines in a .pla. Quite clearly, that is impractical.

either Verilog HDL netlists or decision diagrams, and to add circuit synthesis functionality from such representations to *MustangQ*. Structural Verilog HDL netlists allow for representing much larger functions than `.pla` files can accommodate, because they allow for multi-level specifications that do not require explicitly listing each input/output pair, or even a significant fraction thereof in most cases.²⁴ Similarly, several forms of decision diagram, including BDDs,^{22, 25} can represent bitstring functions in such a way that the function can be efficiently divided into discrete portions. If each of these portions could be efficiently synthesized into reversible circuits and put back together appropriately, then decision diagram representations might allow for inverting even very large hash functions.

To date, we have reversed only small hash functions represented as `.pla` files, because we do not have access to hash functions specified as Verilog HDL netlists of combinational gates nor as decision diagrams. As obtaining such representations is not trivial, future work ought to develop such representations up to and ideally including a 256-bit hash.

Second, temporal parsing and processing constraints. The `.pla` format has served us very well in development of *MustangQ*, as it is human-readable, which not only allows for straightforward development and testing, but also for clear illustration of the processes and optimizations that the toolkit is capable of. Furthermore, it has been incredibly important to have ready functions for testing that are provided in a standardized—or at least standardizable—format. Consequently, we have no plans to abandon `.pla` representation.

However, as has been mentioned several times throughout this article, `.pla` functions have limited spatial efficiency and, in part because of their human-readability, are not efficient to parse or write. Currently, our procedure is implemented with the need to read and write `.pla` files at several points, including to interface with the ESOP minimization step. Alternative data structures—including decision diagrams—should allow for a more direct interfacing that would require fewer computationally-expensive read/write operations. So, although *MustangQ* currently does leverage an internal data structure for representing the content of `.pla` files for as much processing as possible, exploring additional data structures should allow for faster processing times that therefore give a more practical sense of the scalability of our procedure.

REFERENCES

- [1] Paar, C., Pelzl, J., Paar, C., and Pelzl, J., “Hash functions,” *Understanding Cryptography: A Textbook for Students and Practitioners*, 293–317 (2010).
- [2] Preneel, B., “The first 30 years of cryptographic hash functions and the NIST SHA-3 competition,” in [*Cryptographers’ track at the RSA conference*], 1–14, Springer (2010).
- [3] Mironov, I. et al., “Hash functions: Theory, attacks, and applications,” *Microsoft Research, Silicon Valley Campus*, 1–22 (2005).
- [4] Zheng, Z., Xie, S., Dai, H.-N., Chen, X., and Wang, H., “Blockchain challenges and opportunities: A survey,” *International journal of web and grid services* **14**(4), 352–375 (2018).
- [5] Chaudhary, S., Schafeitel-Tähtinen, T., Helenius, M., and Berki, E., “Usability, security and trust in password managers: A quest for user-centric properties and features,” *Computer Science Review* **33**, 69–90 (2019).
- [6] Wohlmacher, P., “Digital certificates: a survey of revocation methods,” in [*Proceedings of the 2000 ACM workshops on Multimedia*], 111–114 (2000).
- [7] Sobti, R. and Geetha, G., “Cryptographic hash functions: a review,” *International Journal of Computer Science Issues (IJCSI)* **9**(2), 461 (2012).
- [8] Forrié, R., “The strict avalanche criterion: spectral properties of boolean functions and an extended definition,” in [*Advances in Cryptology—CRYPTO’88: Proceedings 8*], 450–468, Springer (1990).
- [9] Nielsen, M. A. and Chuang, I. L., [*Quantum computation and Quantum Information*], Cambridge University Press, Cambridge, England, 10th. ed. (2011).
- [10] Henderson, E. R., Henderson, J. M., Sinha, A., Larson, E. C., and Thornton, M. A., “Demonstration of a hardware-independent toolkit for automated quantum subcircuit synthesis,” *arXiv preprint arXiv:2309.01028* (2024).

- [11] Henderson, J. M., Henderson, E. R., Sinha, A., Thornton, M. A., and Miller, D. M., “Automated quantum oracle synthesis with a minimal number of qubits,” in [*Proceedings of SPIE Defense + Commercial Sensing: Quantum Quantum Information Science, Sensing, and Computation XV*], SPIE, Orlando, FL (2023).
- [12] Sinha, A., Henderson, E. R., Henderson, J. M., Larson, E. C., and Thornton, M. A., “A programmable true random number generator using commercial quantum computers,” in [*Proceedings of SPIE Defense + Commercial Sensing: Quantum Quantum Information Science, Sensing, and Computation XV*], SPIE, Orlando, FL (2023).
- [13] Sinha, A., Henderson, E. R., Henderson, J. M., and Thornton, M. A., “Automated quantum memory compilation with improved dynamic range,” in [*Proceedings of the 2022 IEEE/ACM Third International Workshop on Quantum Computing Software (QCS)*], 22–35, IEEE/ACM, IEEE Computer Society, Dallas, TX (November 2022).
- [14] Smith, K. N. and Thornton, M. A., “Quantum logic synthesis with formal verification,” in [*Proceedings of the 62nd International Midwest Symposium on Circuits and Systems*], 73–76, IEEE, IEEE Computer Society, Dallas, TX (August 2019).
- [15] Smith, K. N. and Thornton, M. A., “A quantum computational compiler and design tool for technology-specific targets,” in [*Proceedings of the 46th International Symposium on Computer Architecture*], 579–588, ACM, ACM, Pheonix, AZ (June 2019).
- [16] Preskill, J., “Quantum computing in the nisq era and beyond,” *Quantum* **2**, 79 (2018).
- [17] Castelvechi, D., “IBM releases first-ever 1,000-qubit quantum chip,” *Nature* **624**(7991), 238–238 (2023).
- [18] Chapman, P., “Scaling ionq’s quantum computers: The roadmap,” *IonQ Blog* (2020).
- [19] Fazel, K., Thornton, M. A., and Rice, J. E., “Esop-based toffoli gate cascade generation,” in [*Proceedings of the 2007 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*], 206–209, IEEE, Victoria, CA (2007).
- [20] Brayton, R. K., Hachtel, G. D., McMullen, C., and Sangiovanni-Vincentelli, A., [*Logic minimization algorithms for VLSI synthesis*], Springer Science & Business Media, Berlin/Heidelberg, Germany, 2nd. ed. (1984).
- [21] Rudell, R. L., “Multiple-valued logic minimization for pla synthesis,” Memorandum Report ADA606736, University of California—Berkeley Electronics Research Lab, Berkeley, CA (June 1986).
- [22] Wille, R., Große, D., Teuber, L., Dueck, G. W., and Drechsler, R., “Replib: An online resource for reversible functions and reversible circuits,” in [*Proceedings of the 38th International Symposium on Multiple-Valued Logic*], 220–225, IEEE, IEEE Computer Society, Dallas, TX (May 2008).
- [23] Mishchenko, A. and Perkowski, M., “Fast heuristic minimization of exclusive-sums-of-products,” in [*Proceedings of the 5th International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*], Mississippi State University, Starkville, Mississippi (2001).
- [24] Reese, R. B. and Thornton, M. A., [*Introduction to logic synthesis using Verilog HDL*], Springer Nature (2022).
- [25] Bryant, R. E., “Graph-based algorithms for boolean function manipulation,” *IEEE Transactions on Computers* **C-35**, 677–691, (August 1986).