2013 IEEE Symposium on Security and Privacy

Circuit Structures for Improving Efficiency of Security and Privacy Tools

Samee Zahur and David Evans
University of Virginia
[*samee, evans*]@*virginia.edu*

Abstract—**Several techniques in computer security, including generic protocols for secure computation and symbolic execution, depend on implementing algorithms in static circuits. Despite substantial improvements in recent years, tools built using these techniques remain too slow for most practical uses. They require transforming arbitrary programs into either Boolean logic circuits, constraint sets on Boolean variables, or other equivalent representations, and the costs of using these tools scale directly with the size of the input circuit. Hence, techniques for more efficient circuit constructions have benefits across these tools. We show efficient circuit constructions for various simple but commonly used data structures including stacks, queues, and associative maps. While current practice requires effectively copying the entire structure for each operation, our techniques take advantage of locality and batching to provide amortized costs that scale polylogarithmically in the size of the structure. We demonstrate how many common array usage patterns can be significantly improved with the help of these circuit structures. We report on experiments using our circuit structures for both generic secure computation using garbled circuits and automated test input generation using symbolic execution, and demonstrate order of magnitude improvements for both applications.**

I. Introduction

Generic secure computation protocols and symbolic execution both require arbitrary algorithms to be converted into static circuits, and their efficiency depends critically on the size of the circuit. Therefore, we can improve the speed of these applications by finding efficient circuit constructions for various common programming constructs.

We show efficient constructions for three common data structures: stacks, queues and associative maps. Our constructions are general enough to be used in both the applications. Our stack and queue provide conditional update operations using only amortized $\Theta(\log n)$ gates for each operation, while associative map uses amortized $\Theta(\log^2 n)$ gates for each access or update (where n is the maximum number of elements in the structure). We then show how various common array usage patterns can be rewritten using these data structures, thus obtaining far more efficient circuits for those cases (Section IV). Finally, we demonstrate that the use of these circuits indeed leads to significant speedups in practice (Section VI). We do this by manually replacing standard arrays with our circuit structures in various interesting applications of secure computation and automatic test-input generation (Section V).

In the next section, we present the motivation for our work emphasizing the commonality of static circuits across applications, followed by background on how programs are typically converted into circuits. Section VII discusses related work more broadly.

II. Motivation

There has been a long history of work designing efficient hardware implementations of Boolean circuits, starting with Shannon's work in the 1930s [51]. Hardware circuit designers typically have to worry about circuit depth, gate delay, and power consumption, but view reuse as a design goal. Circuits used in several security applications are quite different. In these applications, each wire in the circuit holds a constant value during the entire execution. This is essential for privacy for secure computation applications, and necessary for test input generation where the goal is to find inputs that lead to a particular output. We call such static-value circuits, *static circuits*. For most applications, including the ones we focus on here, the cost of evaluating a static circuit follows directly from the number of gates in the circuit.

Static circuit structures are radically different from typical hardware circuits. A hardware circuit for adding a million integers, for instance, can fetch them one-by-one from memory, accumulating the sum using a single two-integer adder circuit. Describing the same computation with a static circuit requires a giant structure that includes a million copies of the adder circuit. One particular problem that stems from this difference is that random array access is horrendously expensive in static circuits. Each access of an n-element array requires a circuit of $\Theta(n)$ size where the entire array is multiplexed for the required element by the index being accessed (Figure 1). If the array access is performed in a loop, the corresponding circuit blows up in size extremely rapidly since static circuits cannot be reused. Of course, there are simple cases where this is not a problem, particularly when the access is at positions known in advance

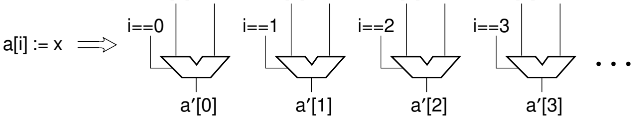
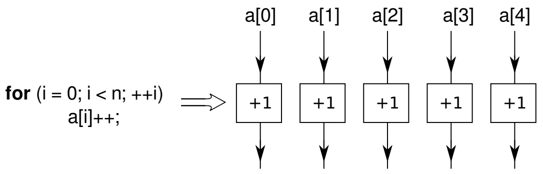


Figure 1. A single array access requiring n multiplexers.

1081-6011/13 $26.00 © 2013 IEEE
DOI 10.1109/SP.2013.40

IEEE computer society



```
for (i = 0; i < n; ++i)
    a[i]++;
```

Figure 2. In this case the index value becomes plain constants once the loop is unrolled. Since the index does not depend on unknown inputs, array access is much cheaper.

(e.g., Figure 2). It is often not the case, however, that all access positions can be determined without knowing the input data. We concentrate on making efficient circuits for the cases in between these extremes: where we know that the array is accessed in some simple pattern, but the indices do depend to some degree on the input data. The overall insight is that most programs do not access arrays in ways that require the general linear multiplexer structure because the actual array accesses are limited in predictable ways. Here, we show how to amortize the cost of multiple accesses when the application either makes multiple accesses that can be performed in a batch, or has some locality in the indices accessed.

In the following subsections, we describe our two target applications: generic secure computation protocols, and automated test-input generation using SAT solvers. We describe their typical use cases, their current state of the art, and how these applications depend on static circuits. Both applications require arbitrary programs to be expressed as static circuits, so efficient circuit constructions yield immediate efficiency gains for both applications.

A. Generic Protocols for Secure Computation

Secure computation allows two (or more) parties to compute a function that depends on private inputs from both parties without revealing anything about either party's private inputs to the other participant (other than what can be inferred from the function output). While there are many application-specific protocols for performing specific tasks securely [15, 42], recent advances in generic protocols enable arbitrary algorithms to be performed as secure computations. To execute any given program securely under such a protocol, it is first converted into a Boolean circuit representing the same computation. After this, the generic protocols specify mechanical ways in which any circuit can be converted into a protocol between parties to perform the same computation securely. The fastest such protocol currently known is Yao's garbled circuits protocol [37, 54]. Recent implementations have demonstrated its practicality for many interesting applications including secure auctions [7, 33], fingerprint matching [30], financial data aggregation [6], data-mining [45], approximate string comparison, and privacy-preserving AES encryption [28].

The static circuits needed for these secure computation protocols do not support fast random access to array elements. This is inherent, since the circuit must be constructed before the index being accessed is known. Any arbitrary array access requires a $\Theta(n)$-sized multiplexer circuit in the general case. While recent work by Dov Gordon et al. [23] has improved the situation for large arrays by with a hybrid protocol using oblivious RAM (ORAM), that approach still has a very high overhead. On the other hand, our approach can be orders of magnitude faster whenever it is applicable, which covers many common cases. (Section VII provides a more detailed discussion.)

Other generic secure computation protocols such as fully homomorphic encryption [17, 53], GMW [21], and NNOB [43] also use static circuit representations of the computation. Therefore our circuit structures are useful in all such protocols, although we only consider garbled circuits in our evaluation.

B. Symbolic Execution on Programs

Another common application of static circuits is in symbolic program execution. Several recent works use symbolic execution to automatically derive properties about program behavior [11, 32, 48]. Several tools are able to analyze legacy programs without requiring any modification to their source code [9, 10, 19].

The particular use of symbolic execution that we consider is automatic test-input generation. The goal here is to analyze a given program and automatically come up with input cases that would drive the program execution along a given path. By exploring all paths to find ones that end in "bad" program states (e.g., a crash or buffer overflow), these tools either obtain concrete test cases that expose program bugs or provide assurance that no such bad paths exist (at least within the explored space).

Test-input generation works by first converting the relevant part of the program into a query for a constraint solver (such as Z3 [12] or STP [16]). This solver is then used to solve for the inputs that will drive program execution to the desired state (or undesired state, as the case may be). Rapid advances in heuristic solvers over the last decade have made it possible to use these tools in many interesting new applications. It turns out that these queries are also equivalent to static circuits [52] in the sense that they also define relationships between variables in a program. Thus, if we can create optimized circuits for programs we also speed up test-input generation, increasing the scale and depth of programs that can be explored.

Since the literature in symbolic execution typically does not refer to circuits, but instead talks solely in terms of constraints, we clarify the relationship between them with an example. Consider the code fragment:

```
x := 5+y; if (x > a) then x := x/a;
```

The goal of the symbolic execution is to check if a division by zero can arise for any particular values of y and a. Normally, the code path to division would be translated into the following constraints: $x=5+y$, $x>a$, $a=0$. If all these constraints can be satisfied for some x, y, and a, we have a possible bug. Solving the constraint is done by feeding it into a SAT solver (as is often needed). The addition and greater-than operations need to be defined using primitive Boolean gates such as AND, OR, etc. much the same way hardware logic gates are used to form addition and comparison circuits. So, whenever we say that the "wires" for p and q are fed into an AND gate to produce the output wire r, what we really mean is that we are adding a new constraint of the form $r = p \wedge q$. This in turn gets translated into $(p \vee \neg r) \wedge (q \vee \neg r) \wedge (\neg p \vee \neg q \vee r)$ which is the input to the SAT solver. Thus, our optimized circuit constructions will be used to produce smaller constraint sets for encoding various programming constructs.

Since arrays can rapidly drive up costs, SMT solvers used in test-input generation tend to put a lot of effort into handling them properly. Some approaches rely on complicated under-approximation strategies that that use a simplified, but less accurate, circuit for quickly discarding obviously unreachable code paths. More accurate circuits are then generated only for the remaining paths. In our evaluation, we do not use any such approximations — instead we generate completely faithful circuits and show how they can be optimized in various cases. We hope that this will enable faster generation of test-inputs by allowing SMT solvers to use fewer, simpler approximation circuits, thereby completing analysis using fewer invocations of computationally expensive SAT solvers. We discuss this further in Section IV-B.

An important characteristic of constraint solvers is that they support cyclic circuits. In the end, their input is just a set of logical constraints on a set of variables. Hence, it is perfectly acceptable to have constraints such as $a = b \vee c$ and $c = a \wedge \neg d$ even though that may seem like circular definition — it is just a set of constraints on the values of the variables a, b, c and d. We will see later that this allows us to optimize random array access in the general case, something we could not do in the case of secure computation.

III. Background

When programs are compiled into static circuits, conversion for most simple statements and conditionals is fairly intuitive. First, statements such as $x := x + 5$ are converted into single assignment form $x_2 = x_1 + 5$, so that each variable is assigned a value only once. This way, we can now allocate separate wires in the circuit to represent the values of x before (x_1) and after (x_2) the assignment.

Conditional branches are done by separately converting each branch into a circuit. At the end of the branch, any variable modified along either path is multiplexed at the end according to the branch condition. For example,

if $(a = 0)$ **then** $x := x + 5$

is converted to

$x_2 := x_1 + 5,\ x_3 := \mathrm{mux}(a = 0, x_1, x_2)$

where $\mathrm{mux}(p, a, b)$ uses its first argument as control bits to select between its second and third arguments.

Such multiplexers are actually never emitted by older symbolic execution tools, since they only explore a program one path at a time. However, this often led them to face the exponential path explosion problem. Modern tools, therefore, often explore paths at the same time using techniques such as path joining [24, 36] and compositional symbolic execution [1], which require such multiplexers (the literature also refers to them as *if-then-else* clauses).

Loops and Functions. Loops and functions pose particular challenges for static circuits. Loops are completely unrolled for some number of iterations, and functions are entirely inlined (see Section VII for details). Since all values are static, and we cannot reuse the same loop body circuit for different iterations. Instead, we instantiate many copies of the same circuit. In test-input generation, there are a wide variety of heuristics for determining an unrolling threshold, which can be as simple as always unrolling just once. In such cases, checking tools simply do not explore paths that require multiple loop iterations, ignoring bugs that depend on such paths [3, 14]. Finding loop bounds is orthogonal to our work, and we do not address it here. In our evaluation, however, we use programs with known loop bounds, and those loops are completely unrolled when converted to circuits.

In secure computation, the practice is a little different. The loop bounds are known based on the given input data size even before the computation begins. An upper bound to input data sizes is publicly revealed even though the data itself is private. For this, they describe circuits in custom languages [25, 39, 50] where inputs known at circuit generation time are treated differently from inputs to the circuit wires itself. Even if the computation is written in a traditional language such as C [26], constraints are placed on what kinds of variables can define loop bounds.

IV. Circuit Structures

In this section we present circuit structures that provide efficient constructions for stacks, queues and associative maps. The stack and queue have quite similar structures, and are therefore discussed together. In each case, we discuss the operations we support, the circuit size required for each, and when these more efficient constructions can be used in place of general arrays.

A. Stack and Queue

We replace arrays with stacks and queues whenever we can determine that the array index only changes in small

```
if (x != 0) {                 t = stk.top();
  a[i] += 3;                  t += 3;
  if (a[i] > 10) i++;   ==>   stk.condModifyTop (x != 0, t);
  a[i] = 5;                   stk.condPush (x != 0 && t > 10, NULL);
}                             stk.condModifyTop (x != 0, 5);
```

Figure 3. Using stacks instead of arrays when program changes i only in small increments. We assumed that the stack top has already moved to the position corresponding to a[i] using condPush during previous increments.

increments or decrements. In other words, we use them to exploit locality whenever possible. Figure 3 shows an example of code transformations required for this. The operations needed to support the transformation are simply *conditional* variants of standard stack and queue operations. Each conditional operation takes an extra Boolean input that either enables or disables the corresponding modification to the stack. So, for example, stk.condPush (c,v) would push v onto the stack if c is **true**. Otherwise, the stack passes through unmodified.

Therefore, we now need to implement such operations efficiently. The operations we support for stack are condPush, condPop, condModifyTop, and readTop (no conditional read is needed since it has no side effects). The queue operations are identical, except that we use the word Front instead of Top. Figure 4 shows a naive implementation of the condPush operation, which still suffers from the expected $\Theta(n)$ cost per operation. We first describe the efficient circuits for stacks; then, we summarize the differences for queues.

In terms of array access patterns, we can use these two structures to optimize any case where the index moves in small increments or decrements. For example, if it is only incremented (or only decremented) in a code fragment, we use the pattern in Figure 3. If it moves in both directions, we just need two stacks "head-to-head", so that a pop from one is matched by a push into the other. If we need multiple array indices, we can use queues. For example, if i and j are both scanning through the same array, the space between them can be modeled as a queue, while the other segments of the array can still be stacks. This can be generalized to multiple index variables in the obvious way by using multiple queues for parts between any two consecutive index variables.

Hierarchical Stack Implementation. The key idea is to split up the stack buffer into several pieces and have empty spaces in each of those, so that we do not have to slide the entire buffer on each operation. This is illustrated in Figure 5. This idea was inspired by the "circular shifts" idea described in Pippenger and Fischer's classic paper on oblivious Turing machines [47]. However, our construction, which we describe now, is significantly modified for our purposes since we do not want to incur the overhead of a general circuit simulating an entire Turing machine. Section VII explains the differences between Pippenger and Fischer's construction and ours in more detail.

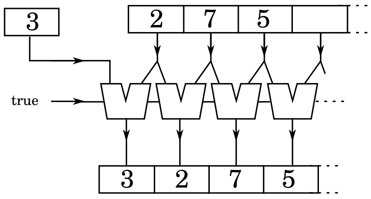


Figure 4. A naïve circuit for condPush, using a series of multiplexers. Since the condition is **true**, a new element is pushed. Had it been **false**, the stack would passed through unmodified.

The buffers of the stack are now in chunks of increasing size, starting with size 5, and then 10, 20, 40 etc. In general, the buffer at level-i has 5×2^i data slots, where the levels are numbered from the top starting at 0. The left side in the figure at level-0 represents the top of the stack. We also maintain the invariant that the number of data slots actually used in the buffer at level-i is always a multiple of 2^i. So, for example, the level-1 buffer only accepts data in *blocks* of two data items. To keep track of the next empty block available, we also maintain a 3-bit counter at each level, t. At any given state, the counter can have values in the range 0–5, and if the one at level-i reads p, then it means that the next data block pushed in to this buffer should go to position p. Finally, we maintain a single-bit "present" flag associated with each data block to indicate whether or not it is currently empty. We do not explicitly show this bit in our figures, other than indicating it with the color of the data block (gray indicates occupied).

Conditional Push Operations. For simplicity, let us start with a "nice" state, where we assume that the counter at

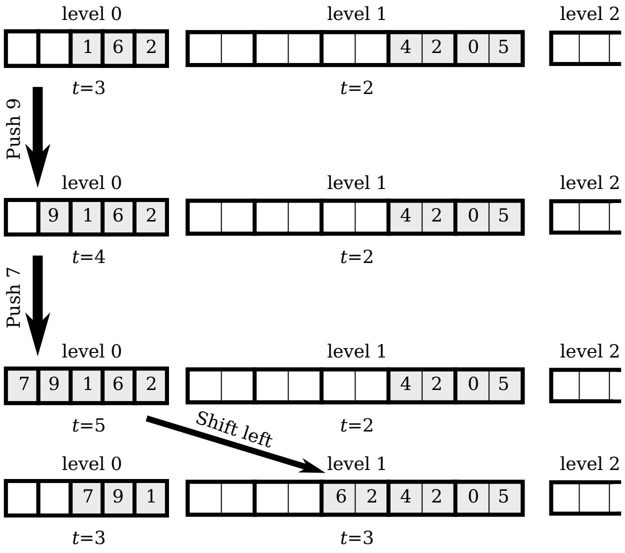


Figure 5. The stack buffers separated into levels, with five blocks each. A level 0 shift circuit is generated after every two conditional push operations.

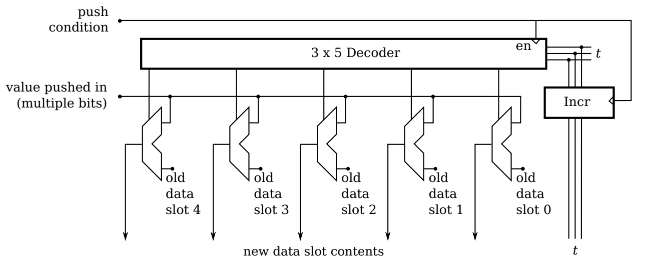


Figure 6. Circuit for a single conditional push operation into level-0 buffer.

each level starts no higher than 3. The top row in Figure 5 depicts such a state. This means every level currently has enough empty slots to accept at least two more blocks of data. So, for the first two condPush operations, we know we have an empty space at level-0 to store the new element as needed if the input condition is **true**. This circuit will simply be the naïve array write operation. In addition, we conditionally increment the counter t, to reflect the change in position of the stack top. This circuit is a series of 5 multiplexers, each of which chooses between the new incoming data being pushed on the stack and the old data stored in the corresponding slot at level-0. The multiplexer control lines are the outputs of a decoder driven by the counter, so that only the appropriate data slot gets written to, while the other items pass through with their values unchanged. Finally, since we are implementing a conditional push, the input condition feeds into the enable bit of the decoder, and conditionally increments the counter at level 0. The wires for the deeper levels (not shown in the figure) are passed through unchanged to the output. This construction is shown in Figure 6.

After two conditional push operations it is possible that the level 0 buffer is now full, and we have to generate some extra circuitry for passing blocks into the next level. For this, we simply check the counter to see if it is greater than 3. The result of this comparison is used to conditionally right-shift the contents of the buffer by 2 slots, while the elements ejected from the right are pushed into level 1 by a conditional push circuit for the deeper level. At the same time, the counter on buffer 0 is decremented by 2 if necessary. Of course, if the counter is already less than or equal to 3 (e.g., if the previous two conditional push operations had false conditions and did not actually do anything), the stack state is not modified in any way and the circuit simply passes on the current values (Figure 7). After all this is done, we can now be sure that the level-0 buffer once again has at least two empty slots for the next two push operations to succeed. The circuit for conditional push into deeper levels is the same as the one for level 0, except that the circuit that shifts to level-i is only generated every 2^i condPush operations.

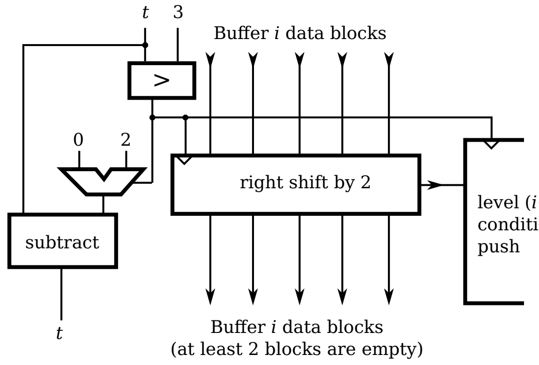


Figure 7. Emptying out data blocks from level i to $(i+1)$.

Conditional Pop. The conditional pop circuit is designed analogously to the conditional push. Normally, it reads from the level-0 buffer and then decrements t. After every 2^i conditional pop operations we add extra circuits to check if the counter at level i is less than 2, and if so, pop 2^{i+1} items from the deeper level (that is, from level $(i+1)$ to level i). Whenever we pop from level i, we decrement the counter at that level, and add 2 to the counter at level $(i-1)$ (unless $i-1<0$). That way, the topmost item on the stack is always in the level-0 buffer. The read top and modify top operations, therefore, just need to use the level-0 counter to determine which of the five elements in the buffer is actually the top, and requires a constant-sized circuit, such as a multiplexer. In this case we do not even need to check all 5 data slots, since some are always kept empty by construction (they are used only in transient states just before a shift).

Since we want to support push and pop operations interleaved arbitrarily, we have to make sure that e.g., a shift from level 0 to level 1 after some conditional push still leaves enough elements at level 0 for any subsequent pops, so that they do not interfere with each other. We only shift two blocks at a time, and only do this when $t \geq 4$, so that after a shift we still have at least two blocks left for any pops that may follow. Similar logic also holds for shift after pop, when we must leave enough empty spaces for push operations, while populating this level for pops. This also explains our choice of using 5 blocks on each level: in the worst case we need 2 empty spaces for push, 2 filled blocks for pop, and one extra space since the deeper levels only take an even number of blocks.

Analysis. From this point we will use the term *cost* of a circuit to mean the number of gates. This is the most important metric for our target applications, and the depth of the circuit is mostly unimportant.

For every two push operations of the stack, a level-1 push circuit is generated only once. This will in turn cause a level-2 push circuit on every four operations of the stack, and so on. In general, for a finite-length stack known to have at most n elements at any given time, k operations access level i at most $\lfloor k/2^i \rfloor$ times. However, the operations

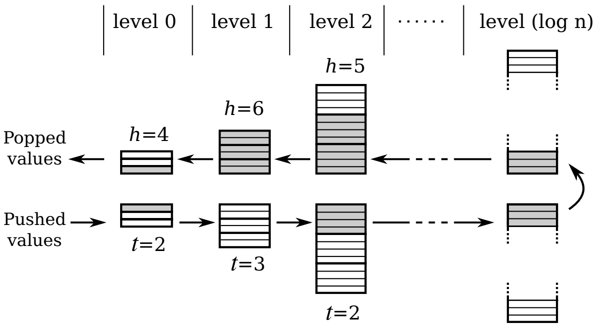


Figure 8. Hierarchical queue construction.

at the deeper levels are also more expensive since they move around larger data blocks — each circuit at level i has $\Theta(2^i)$ logic gates per operation. So when k is large, we generate $\Theta(2^i \times k/2^i) = \Theta(k)$-sized circuits at level i. And since we have $\Theta(\log n)$ levels, the total circuit size for k operations is $\Theta(k \log n)$. Thus, the amortized code for each conditional stack operation is $\Theta(\log n)$. If no upper-bound to the stack length is known in advance (at circuit generation time), we can simply assume that after k conditional push operations, the length is at most k. Following similar reasoning, the amortized cost of a conditional pop is also $\Theta(\log n)$. Operations condModifyTop and readTop only involve the level-0 buffer, and therefore have fixed costs.

Hierarchical Queue Implementation. The queue structure is essentially equivalent to a push-only stack and a pop-only stack, juxtaposed together (Figure 8). Each uses 3×2^i data slots at level i, exactly half the buffer we have that level. The head and tail buffers individually are smaller than in the case of stack (3×2^i instead of 5×2^i) since we know it is either push-only or pop-only. Every level now has two counters, one for head and the other for tail. Each is represented by three bits, with values in the range 0–6, representing the head and tail position of the queue in the current buffer, respectively. If their values are h and t in the buffer for level i, it represents the fact that buffer slots $2^i t, 2^i t + 1, \ldots, 2^i h - 1$ are currently occupied. The invariant $t \leq h$ is always maintained. If $t = h$, it represents the condition where the corresponding buffer is empty. In such cases, we will always reset t and h to the value 3, so that they both point to the middle of the buffer. Here, we are using the convention that pop operations occur at the head, while push operations occur at the tail.

Conditional push and pop operations still work at level-0 as in the stack. After every 2^i push operations, we check level-i and shift two blocks to level-$(i+1)$ if $t < 2$. Similarly, after 2^i pop operations, we resupply the head buffer with new elements from the next level if $h < 5$. So far, this is exactly the same scenario as in the stack. But we now need to add some extra circuitry to transfer elements between the two halves. In particular, when a level-i pop occurs, it is possible that level-$(i+1)$ is empty, or that it does not even exist (that is, we have no wires representing that buffer). So we need to add extra circuits to check for this case. When it occurs, the next few pop operations will supply data straight out of the level-i tail buffer (instead of popping from the empty buffer at the deeper level). Similarly, after a level-i push, if the tail buffer is getting full and the next level is empty, the circuit also checks to see if the head buffer in the current level is also empty. If so, it simply shifts data blocks from the tail buffer directly to the head buffer in the same level, skipping the next-level buffer. All these conditional data movements add extra multiplexers, but increase the circuit-size only by a constant factor. So, we can still provide conditional queue push and pop operations at $\Theta(\log n)$ cost. As we will see later in our evaluation, the constant factors are still quite small, and the total cost of our queue construction is only slightly higher than that of our stack.

Finally, this design ensures that the queue head is always found at the level-0 buffer. So, the read/modify operations for the queue head can still be done at constant cost.

B. Associative Map

The circuits for associative maps are quite different since in this case we cannot rely on any locality of access. Instead, here we amortize the cost whenever we have multiple (read or write) operations that can be performed in a "batch". The only constraint here is that none of the keys or values used in batched operations may depend on the result of another operation in the same batch since this would lead to a cyclic circuit. Many applications do have batchable sequences of array accesses, such as those that involve counting or permuting of array elements (see Section VI for examples). We start with the construction for performing batch writes on an associative map, and then show how it can be tweaked to perform other operations. Here, we define an *associative map* in a circuit as simply a collection of wires representing a set of key-value pairs where the keys all have unique values. We support batched update and batched lookup operations; inserting new values can be done simply by performing updates on non-existent keys.

The circuit for performing a batch of update operations is shown in Figure 9. The circuit takes in two sets of inputs: the old key-value pairs and a sequence of write operations. The write operations in the sequence are themselves also represented as key-value pairs: the key to update and the new value to be written. However, the sequence can have duplicate keys, and the order of writes among those with the same key matters. The output of the circuit is simply the new key-value pairs for the map. The idea is inspired by the set-intersection circuit used by Huang et al. [27]. We first perform a stable sort on all the key-value pairs, which is then passed through a linear scan that marks for removal all but the latest value for each key. Finally, another

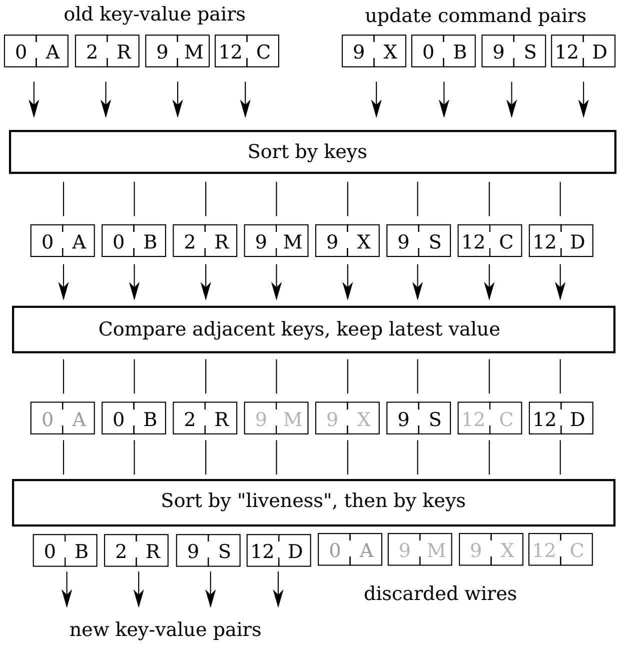


Figure 9. Circuit for batch-updating an associative key-value map.

sort operation is performed, but this time with a different comparison function — this allows us to collect together only the values not marked for removal.

The cost of this cicuit is just the cost of two sorting operations plus a simple linear scan in the middle. Empirically, we found that it works best when the batch size is between approximately n and $2.5n$, where n is the number of key-value pairs in the map. If the batch size gets larger, we can split it up into smaller batches. If the batch is too small, we can still use this method, but the amortized cost may be higher in that case.

The sort operations each require $\Theta(n \log n)$ comparisons, and the linear scan requires $\Theta(n)$ operations. So, the circuit size should be simply $\Theta(n \log n)$[1], providing n operations each with an amortized cost of $\Theta(\log n)$. However, there is a problem. We need an oblivious sorting algorithm, where compare and swaps are hardcoded in the circuit. But, the known efficient oblivious sorting algorithms [4, 22] are not stable — they do not preserve the order of elements in input that compare equal. So to make the sorting stable, we need to pad each element with extra wires feeding them with their sequence number in the input ordering, so that even equal elements no longer compare equal during the sort. The downside of this is that we added a $\Theta(\log n)$ cost to our comparison functions, increasing our amortized cost to $\Theta(\log^2 n)$ per write operation. Obviously, this is not necessary if, in our application, we know all the keys in the write are unique. In that case the entire operation is reduced to the simple union operation for associative maps commonly found in many programming languages.

It is now easy to see how this technique can be used to perform other operations. For example, if we wanted to add to old values instead of overwriting them (e.g., if the values are integers), we just need to replace the linear scan in the middle. Even better, since addition is commutative and associative, we do not need a stable sort, allowing us to construct the complete circuit with just $\Theta(n \log n)$ gates.

If we want to perform read operations, the input will be the map key-value pairs and keys to be read. Assuming we have k keys to be read, they are all padded with sequential serial numbers 0, 1, …, $k-1$, which will be used later in the final sorting operation to order the output wires. But for now, they are all sorted by just the keys as before. The next step will now be filling in values for the requested keys, with a very similar linear scan. Finally, a sorting step reorders the values and presents them in the order in which they were requested (by comparing the extra serial numbers initially padded in), so that we know which output wire corresponds to which requested key. All this requires $\Theta(n \log^2 n)$ logic gates.

In the case of automatic test-input generation, one special observation is that cyclic circuits are allowed in its constraint sets (Section II-B). So we can actually have circuits where some of the input keys or values depend on some of the output wires of the circuit. This allows us to represent arbitrary random array access such as where one write depends on a previous read. In fact, it is quite easy to make small tweaks in the structure described to make a single, unified (but more expensive) circuit that accepts an arbitrary mix of read and write commands to be applied in sequence. Since each circuit can do n operations using just $\Theta(n \log^2 n)$ gates, the amortized cost for each array access now becomes just $\Theta(\log^2 n)$.

V. Implementation

Figure 10 depicts the system we use to implement and evaluate our circuit structures. We start with a program, which is then manually converted into the corresponding circuit generator. For this step, we use a custom circuit

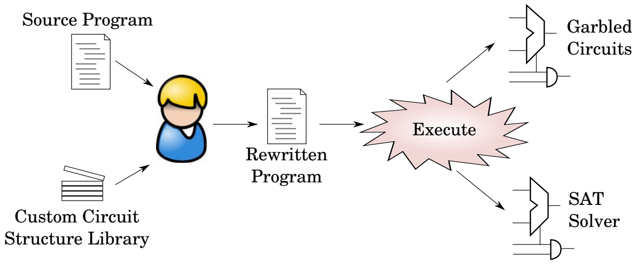


Figure 10. System for testing circuit efficiency.

[1] As we note in Section V, more popular $\Theta(n \log^2 n)$ algorithms actually perform faster for the small data sizes used in our evaluation

```
if (x != 0) {
   a[i] += 3;
   if (a[i] > 10) i++;
   a[i] = 5;
}
```

```
t = stk.top();
t += 3;
stk.condModifyTop (x != 0, t);
stk.condPush (x != 0 && t > 10, NULL);
stk.condModifyTop (x != 0, 5);
```

```
  t ← top stk
  t ← add t (constInt 3)
xnz ← netNot =<< equal x (constInt 0)
stk ← condModTop xnz t stk
 c2 ← netAnd xnz =<< greaterThan t (constInt 10)
stk ← condPush c2 (constInt 0) stk
stk ← condModTop xnz (constInt 5) stk
   ...
```

Figure 11. Steps needed to convert code to circuit. First we simply replace arrays with appropriate data structures whenever possible (Section IV). Then everything is systematically replaced with library calls for generating corresponding circuits e.g. '+' becomes 'add'. Our custom library automatically handles everything internal to the data structure (e.g., condPush automatically decides if it also needs to perform an internal shift etc.). Both these steps are currently done manually (and often combined into a single step).

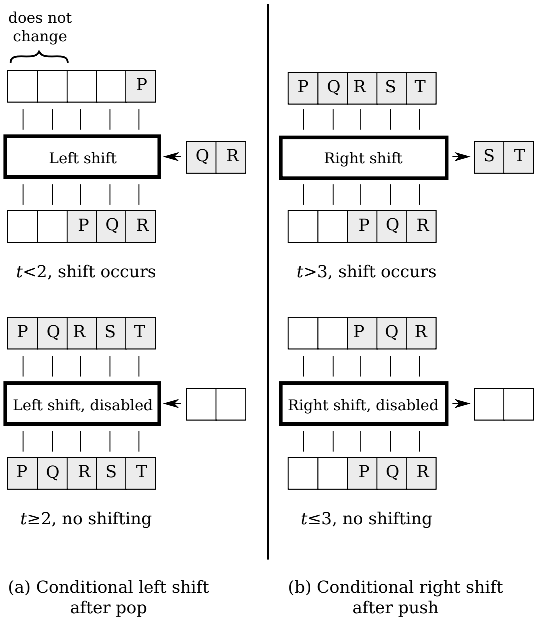


Figure 12. Removing muxes for shifting by determining outputs when generating circuits.

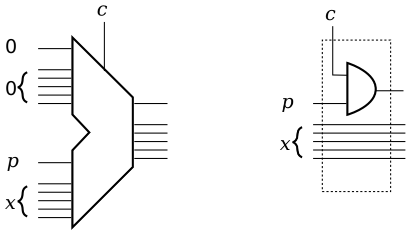


Figure 13. Reducing multiplexers to a single AND gate if one input is known to be blank. If the "present" flag associated with a data block is 0, then the value in the data wires is never used. So, if the blank option is selected by the control bit c, we zero out just the present flag. Note that simple constant propagation would have produced an AND gate for every output wire.

component library written in Haskell that includes all our data structures (Figure 11). This circuit generator is then executed to produce either a description of a secure computation protocol or a SAT query for test-input generation. For secure computation, we generate circuits in the intermediate representation designed by Melicher et al. [40]. For test-input generation, we generate standard DIMACS queries that is accepted by nearly all SAT solvers. We used a fast and popular off-the-shelf solver called Lingeling [5]. The following paragraphs describe various details of the implementation that efficiently realize the designs presented in the previous section. Our implementation and code for all the data structures presented here is available for download from http://mightbeevil.org/netlist/.

Multiplexing in Stacks and Queues. Since most of the data movement in the stack and queue circuits is done by generating multiplexers, they are the most expensive parts of our circuit. Hence, we focused on reducing the number of multiplexers needed.

Consider the conditional shift operations used to move data blocks between consecutive buffer levels. Figure 12 (a) shows what happens when a shift occurs after a pop. Observe that the leftmost two data blocks do not change regardless of whether shifting actually occurred or not. Thus, the input wires can just pass through for these blocks without needing any muxers. We take advantage of similar opportunities for the right shift needed after push operations (Figure 12 (b)), when the output is always a blank block.

This provides substantial benefits by itself, but also enables further reductions that take advantage of knowing blocks are blank at circuit generation time. Figure 13 shows how we can reduce a wide multiplexer into a simple 1-bit AND gate if we know that one of the input data blocks is empty. In secure computation, we have to be careful not to do this if the wires immediately lead to the final output, since this may reveal extra information. Inside our stack and queue constructions, though, this is not a problem. Together, these techniques produce about 28% improvement for the stack circuits, while a more modest 12% for the queue.

Sorting for Associative Maps. Our associative maps require keeping the elements in key-sorted order using a *data-oblivious* sorting algorithm — one where the order of comparison and exchange operations does not depend on the actual values being sorted, since the circuits must be static. Standard sorting algorithms (e.g., Quicksort) are not data-oblivious since the comparisons they do depend on the data. Goodrich's data-oblivious randomized Shellsort algorithm [22] requires $\Theta(n \log n)$ compare and exchange operations. However, the classical algorithm of Batcher's odd-even mergesort [4] produces smaller circuits when we have fewer than about 300,000 elements[2] in the array, even though the latter algorithm has $\Theta(n \log^2 n)$ complexity. All the array sizes we use in our evaluation are small enough for Batcher's algorithm, which is the one we use.

We also take advantage of knowledge about which parts of the input are already known to be sorted. For example, if the associative map is being used as an array, the old data elements (top-left of Figure 9) are already sorted by their index. So, we only sort the command part, and then merge the two sorted parts in a single merge operation. This reduces the overall cost of the batch operation by another 20%. Furthermore, during array operations the keys associated with the old values are just sequential integers that are known at compile time, so basic constant propagation provides another small speedup.

VI. Evaluation

Our evaluation is divided into three parts: first, we compare the size of the our three circuit structures with the ones used in practice for various data lengths; next, we use them in garbled circuits and measure the protocol execution time; finally, we show improvements in test-input generation by automatically producing an input that exposes a buffer overflow bug in an example function. For both secure computation and test-input generation, we were able to obtain at least an order of magnitude speedup for large cases. All the timing measurements were made on a desktop machine running Ubuntu 12.04 on an Intel i7 2600S CPU at 2.8 GHz with 8 GB of memory.

A. Circuit Size Comparison

The graphs in Figure 14 show how the size of our stack and queue circuits scale with increasing data sizes. We report the total number of binary logic gates used.[3] The baseline structures that we compare against are implemented using conventional circuits whose cost scales as $\Theta(n)$ for each operation. We made simple optimizations to the baseline implementation to provide a fair comparison. For example, for the first few operations of a stack, we know that the top of a stack can lie within a small range of indices, and therefore require smaller multiplexers. The stack and queue circuits were generated using random push and pop operations. As expected, the stack and the queue structures have very similar characteristics, and we reduced circuit size by over 11 times for 512 elements. Thus, when converting programs to circuits, these structures can easily replace arrays even for small sizes whenever the access pattern permits.

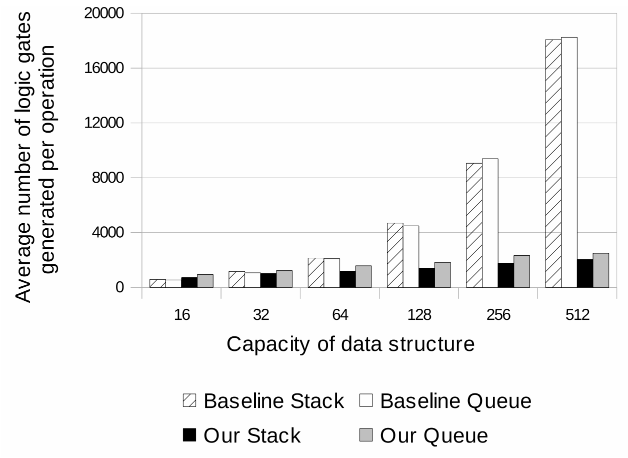


Figure 14. The x-axis shows the maximum capacity of each data structure in number of 16-bit elements. The y-axis is the number of gates.

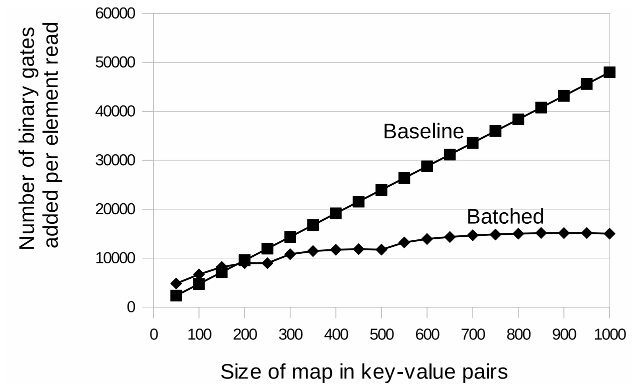


Figure 15. Per-element cost of performing n read operations on an array of n elements. Baseline uses a simple multiplexer, while the "batched" circuit uses an associative map. All maps are integer-to-integer maps, values being 16-bit integers and keys $(\log_2 n)$-bit integers.

Figure 15 shows the benefits of our associative map construction. The baseline in the figure shows the cost of a normal read access by using a multiplexer. We compare that against the size of a batch read circuit on an integer-to-integer associative map. Our structures are worse than the baseline implementation for small sizes, but become beneficial for modest sizes. For 1000-element arrays, our associative map design reduces the circuit size by 3.2*x*. We performed similar experiments for write operations and integer add operations. The trends are quite similar, although batched integer add has much smaller circuits since stable sorting is not required (Section IV-B).

[2] The threshold apparently rises to 1.2 million elements when performing secure computation in the fully malicious model, since using randomized Shellsort then requires an extra shuffle network. Thanks to abhi shelat for pointing this out.

[3] The literature in secure computation often excludes XOR gates in circuit sizes since many protocols, including garbled circuits [34], can be implemented in ways that enable XOR to be computed without any cryptographic operations or communication. While we do not show separate graphs plotting only non-XOR binary gates, we note that they show very similar trends — the *y*-axes simply gets scaled across the board by a factor of approximately one-third.

```
**Input:** $A[i]$ 1st share of $i$-th data item
**Input:** $B[i]$ 2nd share of $i$-th data item
**Output:** $H[d]$, frequency of $d$ in data set
1: $H \leftarrow \emptyset$
2: **for** $i \leftarrow 1, n$ **do**
3:     $H[A[i]+B[i]] \leftarrow 1 + H[A[i]+B[i]]$
```

Figure 16. A simple aggregation of data from secret shares for computing histograms. Output is a simple frequency distribution of the component-wise sums. All elements are 16-bit integers, with sums being modulo-2^{16}.

B. Secure Computation Using Garbled Circuits

Here we demonstrate how much speedup our circuit structures can provide in a garbled circuits protocol execution. For this, we use two simple example programs that we executed in garbled circuits: a simple statistical aggregation program, and a data clustering algorithm.

1) Histogram of Sums: Consider a scenario where companies want to aggregate financial data and generate a report that provides a broad picture of the economy that all the companies can use to make better decisions. However, such data is obviously considered sensitive, and nobody wants to reveal their data to a consortium member who might be from a rival company. This is an example where secure multi-party computation was actually used in practice, as described by Bogdanov et al. [6]. Instead of directly sending their data, they send out cryptographic shares of the data to multiple servers which are then aggregated securely to form a report.

We consider one simple example of such an analysis: histogram generation. Suppose the numbers in the actual data have each been divided into two additive shares A and B, such that the i^{th} data element is simply the sum $A[i]+B[i]$. All we want to do now is compute the frequency of each data element inside a secure computation protocol so that no party learns the actual unaggregated data. The code is shown in Figure 16.

To execute this algorithm as a garbled circuits protocol, the code first needs to be converted into a circuit. In our case, both the arrays are of equal length n. Once the loop is unrolled n times to be converted into a circuit, it is easy to see that i values will all become constants that do not depend on input data. So the only array access that will be slow is the one on line 3 — the sums are not known ahead of time and depend on input values. Since we are performing addition on the elements of H, we can take advantage of our batch element addition circuit here. The speedup we achieve just by making this one single change is shown in Figure 17. For the largest test cases we tried (with $n = 512$), we reduced runtime from 1 minute 18 seconds to just under 7 seconds — a 6.7x speedup.

2) DBSCAN Clustering: Clustering algorithms are often used to uncover new patterns in a given database. For instance, insurance companies could perform clustering to find out how many demographic categories they have in their customer base in order to offer insurance plans accordingly. While it is common for a single company to perform such analyses on their own database, companies might sometimes want a more complete picture by collaborating and running a clustering analysis on their combined databases. Directly sharing such data, however, may not be desirable with rivals, or even possible because of their customer agreements. Thus, it would be desirable to perform this over a secure computation protocol. Here we will show how our stack structure can help construct efficient circuits for a popular such clustering algorithm, DBSCAN [13].

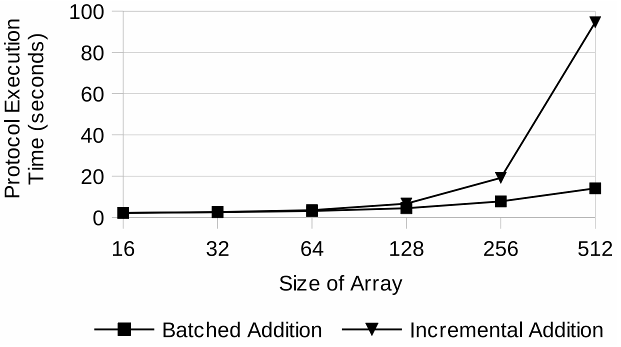


Figure 17. Execution time for the histogram-of-sums protocol for financial data aggregation. All inputs are 16-bit numbers.

The input is simply an array of multi-dimensional data points. Some of its elements come from one party, while the rest from the other. The output is the number of clusters found, and optionally, the cluster assignment of each data point (where $cluster[i] = j$ iff the i^{th} data point was assigned to cluster j). This assignment could then either be directly revealed to the respective parties, or be used in further computation (e.g. to calculate the size, centroid, or variance of each cluster).

The DBSCAN algorithm, shown in Figure 18, is a density-based clustering algorithm that runs a recursive search through the input data set for regions of densely clustered points. If any input point p in the input has at least $minpts$ points within a distance of $radius$, all these points are assigned to the same cluster. The code shown here proceeds in a depth-first search, and has execution time in $\Theta(n^2)$. Efficiently converting it into a circuit, however, poses a number of challenges, which we discuss next.

The first problem we face concerns loop unrolling. Lexically, we see that the code has loops nested up to three levels deep (labelled in the figure as (A), (B) and (C)). Therefore, if we naïvely unroll it, the circuit automatically becomes $\Theta(n^3)$-sized, even though we know only $\Theta(n^2)$ iterations will actually be executed. We know this because loop (B) will be skipped if loop (A) is currently at a point that has already been assigned a cluster (this check occurs at line 6). So, we know that the body of loop (B) is executed

```
Input: $P$: an array of data points
Input: $minpts, radius$
Output: $cluster$: cluster assignment for each point
Output: $c$: number of clusters
1:  $n \leftarrow |P|$
2:  $c \leftarrow 0$
3:  $s \leftarrow$ emptyStack
4:  $cluster \leftarrow [0,0,\ldots]$
5:  for $i \leftarrow [1,n]$ do                                ▷ (A)
6:      if $cluster[i] \neq 0$ then
7:          continue
8:      $V \leftarrow$ getNeighbors$(i,P,minpts,radius)$
9:      if count$(V) < minpts$ then
10:         continue
11:     $c \leftarrow c+1$                                ▷ Start a new cluster
12:     for $j \leftarrow [1,n]$ do
13:         if $V[j] = \textbf{true} \wedge cluster[j] \neq 0$ then
14:             $cluster[j] \leftarrow c$
15:             $s$.push$(j)$
16:     while $s \neq \emptyset$ do                            ▷ (B)
17:         $k \leftarrow s$.pop()
18:         $V \leftarrow$ getNeighbors$(k,P,minpts,radius)$
19:         if count$(V) < minpts$ then
20:             continue
21:         for $j \leftarrow [1,n]$ do                        ▷ (C)
22:             if $V[j] = \textbf{true} \wedge cluster[j] \neq 0$ then
23:                 $cluster[j] \leftarrow c$
24:                 $s$.push$(j)$
```

Figure 18. DBSCAN implementation.

```
while $c_1$ do
    {loopBody1}
    while $c_2$ do
        {loopBody2}

            ⇓

selector ← outerLoop
while selector ≠ outerLoop ∨ $c_1$ do
    if selector = outerLoop then
        {loopBody1}
    if $c_2$ then
        selector ← innerLoop
        {loopBody2}
    else
        selector ← outerLoop
```

Figure 19. Flattening two nested loops into one. This produces smaller unrolled circuits if a strong bound can be obtained for the total number of iterations of the inner loop.

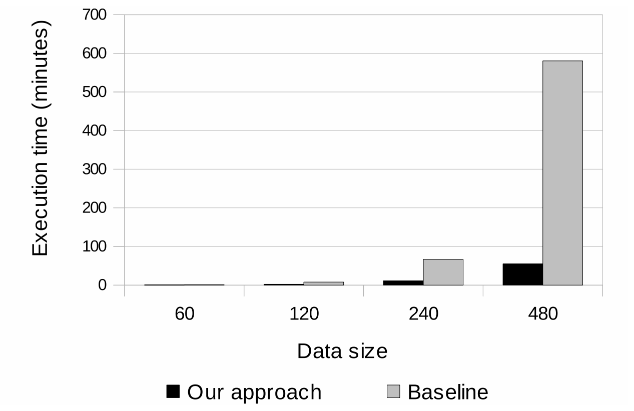


Figure 20. Execution time for DBSCAN clustering protocol over garbled circuits. Data size is in number of data points, where each data point is simply an (x,y) pair of two 16-bit integers, and the distance metric used is Manhattan distance.

at most only n times total, even though it is nested inside another loop. To avoid generating n^2 copies of this loop in the unrolled circuit, we simply *flatten* the loops (A) and (B) as shown in Figure 19. This allows us to unroll the flattened version just $2n$ times. The other loops (e.g., loop (C) or the one at line 12) were not flattened, since we do not have such strong lower bounds on how often they are skipped. Generally, such flattening tends to help only if the number of iterations taken by the inner loop can be substantially different for each iteration of the outer loop, depending on the private inputs.

However, this comes at a small additional cost: the value of the variable i can no longer be determined at the time of circuit generation. So, the array accesses at line 6 and line 8 will now be expensive, requiring full multiplexers. This does not occur at the innermost loop levels, and the asymptotic complexity is therefore unaffected. Most other array accesses in the program involve indices known at the time of circuit generation, and are therefore trivially implemented (e.g., unrolled version of loop (C) will have constant values for j in each copy of the loop body).

The only remaining trouble will be the stack push operation inside loop (C). Without the use of our stack construction, there is no simple way of avoiding yet another $\Theta(n)$ complexity factor here. Simply substituting a naive construction of the stack with our data structure reduces the complexity of the generated circuit from $\Theta(n^3)$ to $\Theta(n^2 \log n)$. The effect of this one simple change is shown in Figure 20. All other optimizations are identical in the compared versions to isolate the impact of just using our stack circuit constructions. In the case of 480 data points, the runtime drops from almost 10 hours to less than 1 hour.

C. Test Input Generation

To evaluate the impact of our structures on symbolic execution, we use the merging procedure of the merge sort

```
1 #define MAXSIZE 100

  int merge (int *arr1, int *arr2, int n, int *dest) {
    int i, j, k;
    if (n > MAXSIZE) return -1;
6   for (i = 1; i < n; ++i)
      if (arr1[i-1] > arr1[i]) return -1;
    for (j = 1; j < n; ++j)
      if (arr2[j-1] > arr2[j]) return -1;
    i = j = 0;
11  for (k = 0; k < 2 * n; ++k) {
      if (arr1[i] < arr2[j]) dest[k] = arr1[i++];
      else dest[k] = arr2[j++];
    }
    return 0;
16 }

  int main() {
    int a[MAXSIZE], b[MAXSIZE], dest[2*MAXSIZE];
    int n;
21  fromInput (a, b, &n);
    merge (a, b, n, dest);
  }
```

Figure 21. A C program fragment for the merge procedure of merge sort. It has a bug since it does not check if i or j are already out of bounds in the last loop body.

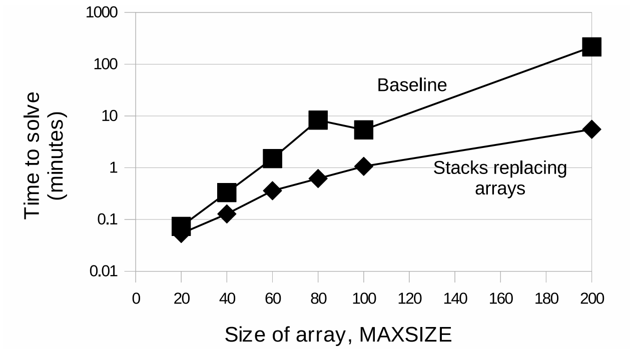


Figure 22. Time taken to solve for an input that triggers a buffer overflow. The scale for the y-axis is logarithmic. For the largest case, the speedup is over $30x$.

algorithm. A version of this code is shown in Figure 21. However, the code shown has a bug: in the last loop, it uses the array elements without first checking if the index is already out of bounds. While the bug is quite simple, most popular automated tools today have a hard time detecting this bug. This is because of the well known path explosion problem, where the number of possible paths that can be taken through the code is exponential in MAXSIZE. We tried using a state-of-the-art symbolic execution tool, KLEE [9], on this example. However, it simply enumerates every single one of these paths, creating a new constraint set for each of them. As a result, it never actually generates a path that exposes the bug.

Instead, we converted the entire computation into a circuit (which is the same as fully joining all the paths together into a large constraint, as described in Section III), and added constraints to let a SAT solver find an input that exposes an out-of-bounds access. We solve the same problem through the SAT solver twice: in one run we change the array access on lines 12 and 13 to use our stack structure, while in the other case we leave it unchanged. The timing results are shown in Figure 22. The circuit structures reduce the time to find the bug from over 3 hours to just 6 minutes when the array size is 200 elements.

In practice, programs often have buffer overflow errors like these that are not triggered unless the data size is large and has a particular pattern of values. However, most complicated access patterns involving such large arrays are considered impractical for symbolic execution systems in use today. Our example here clearly shows the value in thinking of the constraint sets in terms of static circuits, and using that abstraction to create more optimized queries.

Although in our experiments we manually wrote a program that generates queries for this particular function, we expect that this process can be more automated in future. At least for the common cases described in Section IV, it should not be too hard for an automated tool to replace array uses with the stack and queue structures that we describe here, obtaining the same speedups we show here.

VII. Related Work

While we include cyclic circuits in our notion of static valued circuits, the special case of acyclic circuits have been extensively studied in the past. Classical results from circuit complexity provides bounds for the sizes of many interesting families of functions. Examples of such families include symmetric functions [31], monotonic functions [2], and threshold functions [8]. Such functions tend to be too simple or restrictive for our purposes. The result most relevant to our work is Pippenger and Fischer's classical paper on oblivious Turing machines [47], where they show how any Turing machine with sequential access to a tape can be simulated in a combinatorial circuit of $\Theta(n \log n)$ size where n is the number of steps to simulate. While a sequential tape can be immediately used as a stack, it is not obvious how to extend their result to a FIFO queue, which is why we do not use their circuit. Moreover, we optimize our designs to better suit our application targets. For example, Pippenger and Fischer's design needs to support only one general operation — simulating a single Turing machine time step. This is far too general for our needs, and we generate much less expensive specialized push and pop circuits.

Besides results for circuit complexity, the two application targets that we focus on in this paper have seen rapid improvements in speed in recent years. Below we summarize some of the important ideas that made this possible, and how they relate to our work.

A. Secure Computation — Garbled Circuits

Much of the recent work enabling fast execution of garbled circuits actually work at the level of the underlying protocol, and does not necessarily change the circuit being simulated. This includes the free XOR trick [34] that almost eliminates the cost of executing XOR gates, garbled row reductions that reduce communication overhead by 25% [46], and pipelined execution that overlaps the various phases of execution for scalability and reduced latency [28]. Since our techniques do not depend on any of the specifics of the underlying protocol, they can be used in combination with all of these popular optimizations.

While traditionally most garbled circuit execution systems were considered practical only against an *honest-but-curious* adversary, execution of billion gate circuits against a *fully malicious* adversary has also been demonstrated recently by Kreuter et al. [35] by extensive use of parallelism. They used the cut-and-choose technique outlined by Lindell and Pinkas [38] to make their execution resistant against malicious parties. Another, more efficient, technique for fully secure execution of garbled circuits that have been proposed recently by Huang et al. is based on dual execution of the protocol [29, 41]. Since our optimization only impact the circuits, they can be used with any of these flavors of garbled circuit protocols.

Gordon et al. [23] recently demonstrated a hybrid secure computation protocol involving garbled circuits and oblivious RAM that provides general random access to arrays in sublinear time. However, their protocol still has a very high overhead due to the use of Oblivious RAM (ORAM) structures. Our purely circuit-based solutions, on the other hand, are much faster whenever they are applicable. In terms of absolute performance, the fastest they reported was about 9.5 seconds per element access, and that for an array of 2^{18} elements. Because of such high overheads, it was actually still faster to naively multiplex over the entire array unless the array is big (in their implementation, they break-even at arrays of approximately 260,000 elements). Our circuit-structures are orders of magnitude faster in the cases we can handle, and as shown in Section VI we breakeven for much smaller data sizes.

Finally, many of the recent frameworks and compilers such as Fairplay [39] and CMBC-GC [26] provide ways to produce garbled circuit protocols starting from high-level programs. Although we have not focused on automatic circuit compilation in this work, we hope that in future such tools will be able to automatically detect which circuit structure is applicable a given situation. This would allow programmers to write code naturally using standard data structures like arrays, but generate circuits that automatically implement array accesses using appropriate less expensive data structures to achieve reasonable performance.

B. Symbolic Execution

The development of symbolic execution as a tool for static analysis has, in large part, been aided by the concurrent improvements in constraint solvers. With modern constraint solvers, symbolic execution systems such as KLEE [9] and EXE [10] are able analyze a program and solve for inputs that drive execution of the given program along a certain path. We hope that our methods would make it easier for such tools to handle far more complicated programs than they are currently able to.

More recent advances in this field mostly focused on improving scalability of these tools and on solving the exponential path explosion problem. Path explosion is a notorious problem where the number of paths to be explored is exponential in the number of branches along that path. Solutions recently proposed include compositional methods and state merging. Compositional approaches (e.g., [1, 18, 20]) attempt to keep the paths shorter by analyzing one function at a time and composing the results together later, often lazily. In practice, however, state joining [24, 36] seems to provide better results where several paths are merged into one larger query the way we did here. Some researchers have also noted how this method can be seen as a strict generalization of the compositional methods [36].

Something we did not delve into in this paper is how to determine loop bounds. Since current systems require loops to be completely unrolled, tools need to establish a reasonable upper bound for how many iterations of the loop should be explored. There has been a lot of recent work in this area, all of which complements our work on arrays. For example, Obdržálek and Trtík [44] recently showed how integer recurrence equations can sometimes be used to solve for an upper bound. Saxena et al. demonstrated how using input grammar specification can sometimes help determine loop bounds [49]. Given how loops are often used in conjunction with arrays, we believe our work will further broaden the scope of their techniques to more complicated coding patterns.

VIII. Conclusion

We showed how a common set of ideas can be used to speed up common programming patterns in generic secure computation and symbolic execution of programs, two previously unrelated applications. Both of these applications depend on static circuits. We devised circuit structures that lead to large speedups for several common data structures. Further, we demonstrated how slow array access operations in programs can often be replaced by more specialized data structures like stack, queue, and associative map, achieving asymptotic improvements in runtime. We are optimistic that similar approaches can be taken with other data structures to provide similar gains for a wide range of applications. Although our work focused on manual implementation, we believe such transformations could be automated in many cases, and our results point to future opportunities for automatically generating efficient static circuits for secure computation and symbolic execution.

Acknowledgments

We thank William Melicher for developing the backend tools we used for testing our garbled circuit applications. Frequent discussion with Benjamin Kreuter provided valuable resources and citations that helped shape this work in its early stages. We thank abhi shelat for helping us find several further opportunities for optimizations, and thank Yan Huang, Gabriel Robins, and Westley Weimer for their helpful comments and suggestions on this work. This work was supported by grants from the National Science Foundation, Air Force Office of Scientific Research, and DARPA. The contents of this paper, however, do not necessarily reflect the views of the US Government.

References

[1] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann, "Demand-driven compositional symbolic execution," *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[2] A.E. Andreev, "On a method for obtaining lower bounds for the complexity of individual monotone functions," in *Soviet Math. Dokl*, vol. 31, no. 3, 1985.

[3] Domagoj Babic and Alan J. Hu, "Calysto: Scalable and precise extended static checking," in *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*. IEEE, 2008.

[4] Ken E. Batcher, "Sorting networks and their applications," in *Proceedings of the April 30–May 2, 1968, spring joint computer conference*. ACM, 1968.

[5] Armin Biere, "Lingeling and friends at the sat competition 2011," *FMV Report Series Technical Report*, vol. 11, no. 1, 2011.

[6] Dan Bogdanov, Riivo Talviste, and Jan Willemson, "Deploying secure multi-party computation for financial data analysis," *Financial Cryptography and Data Security*, 2012.

[7] Peter Bogetoft, Dan Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Nielsen, Jesper Nielsen, Kurt Nielsen, Jakob Pagter *et al.*, "Secure multiparty computation goes live," *Financial Cryptography and Data Security*, 2009.

[8] Ravi B. Boppana, "Amplification of probabilistic boolean formulas," in *Foundations of Computer Science, 1985., 26th Annual Symposium on*. IEEE, 1985.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008.

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler, "Exe: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, 2008.

[11] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé, "Using symbolic execution for verifying safety-critical systems," *ACM SIGSOFT Software Engineering Notes*, vol. 26, no. 5, 2001.

[12] Leonardo De Moura and Nikolaj Bjørner, "Z3: An efficient smt solver," *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[13] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proceedings of the 2nd International Conference on Knowledge Discovery and Data mining*, vol. 1996. AAAI Press, 1996.

[14] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata, "Extended static checking for java," in *ACM Sigplan Notices*, vol. 37, no. 5. ACM, 2002.

[15] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas, "Efficient private matching and set intersection," in *Advances in Cryptology-EUROCRYPT 2004*. Springer, 2004.

[16] Vijay Ganesh and David Dill, "A decision procedure for bit-vectors and arrays," in *Computer Aided Verification*. Springer, 2007.

[17] Craig Gentry, "A fully homomorphic encryption scheme," Ph.D. dissertation, Stanford University, 2009.

[18] Patrice Godefroid, "Compositional dynamic test generation," in *ACM SIGPLAN Notices*, vol. 42, no. 1. ACM, 2007.

[19] Patrice Godefroid, Michael Y. Levin, and David Molnar, "Automated whitebox fuzz testing," in *Network and Distributed System Security Symposium (NDSS)*, 2008.

[20] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and Sai Deep Tetali, "Compositional may-must program analysis: unleashing the power of alternation," in *ACM Sigplan Notices*, vol. 45, no. 1. ACM, 2010.

[21] Shafi Goldwasser, Silvio M. Micali, and Avi Wigderson, "How to play any mental game, or a completeness theorem for protocols with an honest majority," in *Proc. of the Nienteenth Annual ACM STOC*, vol. 87, 1987.

[22] Michael T. Goodrich, "Randomized shellsort: A simple oblivious sorting algorithm," in *Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms*. Society for Industrial and Applied Mathematics, 2010.

[23] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis, "Secure two-party computation in sublinear (amortized) time," in *19th ACM Conference on Computer and Communications Security (CCS)*, 2012.

[24] Trevor Hansen, Peter Schachte, and Harald Søndergaard, "State joining and splitting for the symbolic execution of binaries," in *Runtime Verification*. Springer, 2009.

[25] Wilko Henecka, Stefan Kögl, Ahmad-Reza Sadeghi, Thomas Schneider, and Immo Wehrenberg, "Tasty: tool for automating secure two-party computations," in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010.

[26] Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith, "Secure two-party computations in ansi c," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012.

[27] Yan Huang, David Evans, and Jonathan Katz, "Private set intersection: Are garbled circuits better than custom protocols," in *Network and Distributed System Security Symposium (NDSS)*, 2011.

[28] Yan Huang, David Evans, Jonathan Katz, and Lior Malka, "Faster secure two-party computation using garbled circuits," in *USENIX Security Symposium*, 2011.

[29] Yan Huang, Jonathan Katz, and David Evans, "Quid-pro-quotocols: Strengthening semi-honest protocols with dual execution," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012.

[30] Yan Huang, Lior Malka, David Evans, and Jonathan Katz, "Efficient privacy-preserving biometric identification," in *Network and Distributed System Security Symposium*, 2011.

[31] V.M. Khrapchenko, "The complexity of the realization of symmetrical functions by formulae," *Mathematical Notes*, vol. 11, no. 1, 1972.

[32] Sarfraz Khurshid, Corina PĂsĂreanu, and Willem Visser, "Generalized symbolic execution for model checking and testing," *Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[33] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider, "Improved garbled circuit building blocks and applications to auctions and computing minima," *Cryptology and Network Security*, 2009.

[34] Vladimir Kolesnikov and Thomas Schneider, "Improved garbled circuit: Free xor gates and applications," *Automata, Languages and Programming*, 2008.

[35] Ben Kreuter, abhi Shelat, and Chih-hao Shen, "Billion-gate secure computation with malicious adversaries," in *Proceedings of the 21st USENIX conference on Security symposium*. USENIX Association, 2012.

[36] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*. ACM, 2012.

[37] Yehuda Lindell and Benny Pinkas, "A proof of security of yaos protocol for two-party computation," *Journal of Cryptology*, vol. 22, no. 2, 2009.

[38] ——, "Secure two-party computation via cut-and-choose oblivious transfer," *Theory of Cryptography*, 2011.

[39] Dahlia Malkhi, Noam Nisan, Benny Pinkas, and Yaron Sella, "Fairplaya secure two-party computation system," in *USENIX Security Symposium*, 2004.

[40] William Melicher, Samee Zahur, and David Evans, "An intermediate language for garbled circuits (poster abstract)," IEEE Symposium on Security and Privacy, 2012, May 2012.

[41] Payman Mohassel and Matthew Franklin, "Efficiency tradeoffs for malicious two-party computation," *Public Key Cryptography-PKC 2006*, 2006.

[42] Moni Naor, Benny Pinkas, and Omer Reingold, "Distributed pseudo-random functions and kdcs," in *Advances in CryptologyEUROCRYPT99*. Springer, 1999.

[43] Jesper B. Nielsen, Peter S. Nordholt, Claudio Orlandi, and Sai S. Burra, "A new approach to practical active-secure two-party computation," *Advances in Cryptology–CRYPTO 2012*, 2012.

[44] Jan Obdržálek and Marek Trtík, "Efficient loop navigation for symbolic execution," *Automated Technology for Verification and Analysis*, 2011.

[45] Benny Pinkas, "Cryptographic techniques for privacy-preserving data mining," *ACM SIGKDD Explorations Newsletter*, vol. 4, no. 2, pp. 12–19, 2002.

[46] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams, "Secure two-party computation is practical," *Advances in Cryptology–ASIACRYPT 2009*, 2009.

[47] Nicholas Pippenger and Michael J. Fischer, "Relations among complexity measures," *Journal of the ACM (JACM)*, vol. 26, no. 2, 1979.

[48] Corina S. Psreanu, Peter C. Mehlitz, David H Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM, 2008.

[49] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song, "Loop-extended symbolic execution on binary programs," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009.

[50] Axel Schropfer, Florian Kerschbaum, and Gunter Muller, "L1-an intermediate language for mixed-protocol secure computation," in *Computer Software and Applications Conference (COMPSAC), 2011 IEEE 35th Annual*. IEEE, 2011.

[51] Claude Shannon, "A symbolic analysis of relay and switching circuits," MIT Master's Thesis, 1937.

[52] G.S. Tseitin, "On the complexity of derivation in propositional calculus," *Structures in Constructive Mathematics and Mathematical Logic, Part II*, 1968.

[53] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan, "Fully homomorphic encryption over the integers," *Advances in Cryptology–EUROCRYPT 2010*, 2010.

[54] Andrew C. Yao, "Protocols for secure computations," in *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, 1982.