It Doesn't Have to Be So Hard: Efficient Symbolic Reasoning for CRCs

Vaibhav Sharma
University of Minnesota
vaibhav@umn.edu

Navid Emamdoost
University of Minnesota
navid@cs.umn.edu

Seonmo Kim
University of Minnesota
kimx3472@umn.edu

Stephen McCamant
University of Minnesota
mccamant@cs.umn.edu

***Abstract*—Cyclic redundancy check functions (CRCs) are commonly used as checksums in hardware and software systems. Though some previous work has treated CRCs as if they would be difficult for automated reasoning or proposed using them for obfuscation, we argue that CRCs need not be difficult for symbolic execution. CRCs have a strong algebraic structure, and the key to efficient reasoning about them is exposing this structure. Common CRC implementation techniques use per-bit branching or lookup tables, but these code structures can be compactly summarized by symbolic execution engines that perform path merging or create formulas reflecting the contents of a table, respectively. To evaluate the effectiveness of these techniques, we test the binary symbolic execution tools angr and FuzzBALL, and the Java bytecode tool Java Ranger, on symbolically finding CRC preimages for output sizes of 32 and 64 bits and input sizes up to 8192 bytes. The best configurations can find short CRC preimages in just a fraction of a second and long ones in under 10 seconds, demonstrating that constraint solving with CRCs is not difficult.**

I. Introduction

Cyclic redundancy check functions (commonly "CRCs") are one of the most commonly used error-detection and checksum functions used in both hardware and software, including in common file formats such as PNG images and bzip2 compression. In cryptography, it is now well understood that CRCs are not resistant to adversarial modification and should not be used in place of cryptographic hash functions or MACs, though insufficient appreciation of this property in the past contributed to weaknesses in WEP [5] and SSHv1 [15]. Hash functions generally and sometimes CRCs in particular have been used as examples of program functionality that presents an obstacle to fuzzing and symbolic execution systems, for instance in motivating approaches that treat such functions as uninterpreted [16] or evaluate them separately [8], [28]. Recently Jung et al. proposed using CRCs to create hard predicates to impede symbolic execution and make it more difficult to discover vulnerabilities. Given that CRCs are not considered difficult to deal with in some other contexts, it is unclear how much long-term difficulty they present.

We investigate this question and argue that CRCs should not be considered difficult for symbolic execution. Common implementation techniques for CRCs include branches or table lookups that can cause path explosion in baseline symbolic execution techniques. But the underlying linear structure of CRCs allows for efficient reasoning in SMT solvers as long as it is made explicit to them. We find that by applying either previously-known path-merging enhancements or a new encoding of table contents, we can easily expose the linear structure of a CRC computation in SMT constraints. We then also confirm that off-the-shelf SMT solvers can efficiently reason about these constraints, including by using the linearity of XOR to perform Gaussian elimination at the level of Boolean constraints.

As a representative symbolic reasoning task using CRCs, we consider what is in essence a pre-image computation. We apply an implementation of CRC computation to an unconstrained symbolic input buffer of a given size, and then have the symbolic execution tool consider whether it is feasible that the CRC of that buffer can have a particular concrete value that we have chosen at random (in fact as the CRC of a random string of the same size). In all cases this condition is feasible, and in the course of verifying that it is, the SMT solver used by the symbolic execution engine produces a satisfying assignment which indicates a value for the buffer contents which has the desired CRC. Such an experiment is similar to the constraints that might arise when symbolic execution is used to create a test case or proof-of-vulnerability for a program whose input format uses a CRC as a checksum. The constraints relating the input to its CRC would be the same, and then there would be other constraints relating to the vulnerability. These experiments are also similar to the task of determining how to trigger a branch which has been obfuscated by changing a simple equality comparison into a comparison of CRC results, as has been proposed by Jung et al. to frustrate hybrid fuzzing.

To match the most practically important uses of CRCs in software, our evaluation uses standard 32-bit and 64-bit CRC functions. Our primary focus is on tools for binary-level symbolic execution, specifically the open-source systems angr and FuzzBALL, taking advantage of their existing implementations of path merging and lookup tables respectively. We also illustrate path merging using Java Ranger, a symbolic execution tool for Java bytecode. We use the symbolic execution tools together with the open-source SMT solvers Z3, Yices, and Boolector (and their underlying SAT solvers). The artifacts and corresponding driver code for each symbolic execution engine are available at https://github.com/navidem/symbolicCRC.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23xxx
www.ndss-symposium.org

II. Background

Next we give a few details about CRC functions and their common software implementations, to explain the features that may or may not be challenging for symbolic execution.

CRCs have in common with other commonly used checksums that they are computed as the remainder after division where the dividend is derived from the input. But they differ in the way that remainder is computed. Mathematically, CRCs are most easily described by thinking of binary data as "bit vectors" in a literal sense: namely as vectors from a vector space whose underlying field is the set $\{0, 1\}$. The integers 0 and 1 form a finite field when XOR is used as the additive operator and AND is used as multiplication; this finite field interpretation of bits is called "GF(2)", the Galois field of two elements. CRCs are computed by treating bit vectors as formal polynomials, and then performing polynomial division with a fixed divisor polynomial that is also called a generator polynomial. All the coefficients are 0s and 1s interpreted as a finite field; for instance since both 0 and 1 are their own additive inverses in GF(2), the subtraction used in long division can also be implemented as bitwise XOR.

The strong algebraic structure of CRCs gives them many properties that ease implementing and reasoning about them. The most important of these is linearity with respect to the XOR operation: analogous to the relationship between addition and remainder operators over integers, the CRC-style remainder of two values combined with XOR is equal to the XOR of the remainder of the two values separately. (To be precise, common CRCs incorporate slight modifications to avoid having the CRC of an all-0-bits vector being 0; this technically makes them "affine" instead of "linear".)

The GF(2) polynomial remainder operations used in CRCs are not directly supported by most instruction sets, so efficient CRC implementations take advantage of their algebraic structure to express the function in different ways in terms of other machine operators. The most direct approach is to consider each input bit separately. Because of linearity, the way the CRC value is updated based on a given input bit varies by either XORing in, or not, a particular value. This leads to a basic loop structure in which for each input bit, a value is conditionally combined using XOR (an example is shown later in Listing 1).

Another common technique used to compute CRCs in software is to pre-compute all the required changes to the running CRC for a given input byte, and then at runtime to determine the value to be combined with the CRC by looking up each input byte in a 256-entry lookup table. In this approach the linearity property of the CRC carries over into the structure of the lookup table contents.

III. Related Work

Several previous applications of fuzzing and symbolic execution have proposed specialized techniques for dealing with checksums, since checksums can generate many dependencies across the input and make test case generation more challenging. One useful technique is to recognize and separate out a checksum computation, and then recompute a checksum after the rest of the input has been chosen [8], [28]. This takes advantage of the fact that checksums are still straightforward to compute in the forward direction. Most recently the REDQUEEN fuzzing system [2] has demonstrated how to apply such a structured approach efficiently using only concrete execution. Another technique that can be used in symbolic execution is to abstract checksums as uninterpreted functions [16]. Though our work shares a similar motivation, our approach is limited to CRCs and argues that they do not need to be treated separately: with appropriate symbolic execution capabilities they can be treated the same as other parts of a program.

One class of enhancements to symbolic execution we take advantage of, which we generically refer to as path merging, has been widely explored as a general way to reduce path explosion. State-merging approaches to path merging apply to symbolic execution systems that represent multiple execution states simultaneously, and opportunistically merge state objects with the same control-flow location [17], [19]. The MultiSE [22] approach also achieves a similar effect by a more drastic restructuring of the symbolic state representation using BDDs. Avgerinos et al. [3] proposed a different approach they dubbed "veritesting" in which a region of code containing branching is translated statically. More recently an extended version of this approach has also been effective for Java [25], [24], [23].

The other class of techniques we use, efficient encodings of the contents of lookup tables, has been developed in previous binary symbolic execution systems, though individual techniques are simple enough that we are not aware they have been the main topic of publications. Cha et al.[9] describe techniques used in their Mayhem system, which include balanced binary trees to cover a set of feasible values, and a formula encoding of adjacent values that satisfy a linear relationship. Too many other optimizations for symbolic execution have been proposed to cover here, but we recommend a recent survey by Baldoni et al. [4] for interested readers.

An interesting recent application of CRCs has proposed them explicitly with the aim of their being difficult for symbolic execution. Jung et al. [18] propose a collection of techniques to thwart fuzzers in discovering vulnerabilities. One such technique targeted symbolic execution; Jung et al. call it AntiHybrid and motivate it as targeting hybrid fuzzing systems such as Driller [27] and QSym [29] that combine symbolic execution with fuzzing to get the fastest test generation performance. Hybrid fuzzers typically fall back on symbolic execution for branch conditions that are difficult to trigger with random mutation; a simple example is a comparison with a large constant "magic number". Such a comparison would normally be quite easy for symbolic reasoning, but Jung et al. propose to obfuscate the check by transforming both sides of the comparison with a CRC. Jung et al. report that such obfuscation lead to timeouts or path explosion when QSym attempted to explore it, and that QSym then heuristically gave up on the branch as not worth exploring. This would prevent the hybrid fuzzer from exploring code reachable only by the obfuscated branch, and in end-to-end experiments Jung et al. found that the number of crashes found by QSym was greatly reduced.

IV. Techniques

Symbolic branch conditions which have both sides feasible commonly cause path explosion in symbolic execution. Such branch condition occur frequently in code, thereby creating a need for symbolic executors to avoid branching when possible. We present the CRC32 implementation used by Jung et al. in Listing 1. The symbolic branch condition on line 10 causes path explosion since the symbolic executor has to branch once for every bit in the input. This branch creates a multiplicative factor of 8 in the number of execution paths to be explored for every byte in the input. Path merging can easily summarize both sides of the branch in the multi-path code region spanning lines 10-12 into the `crc` local variable.

```
1  unsigned int crc32(unsigned char *message) {
2    int i=0, j;
3    unsigned int byte, crc = 0xFFFFFFFF;
4    while (message[i] != 0) {
5      byte = message[i];              // Get next byte
6      byte = reverse(byte);           // 32-bit reversal
7      for (j = 0; j <= 7; j++) {     // Do eight times
8        // the following branch causes path explosion
9        // without path-merging
10       if ((int)(crc ^ byte) < 0)
11         crc = (crc << 1) ^ 0x04C11DB7;
12       else crc = crc << 1;
13       byte = byte << 1;            // Ready next msg
     bit.
14     }
15     i = i + 1;
16   }
17   return reverse(~crc);
18 }
```

Listing 1. CRC32 implementation used by Jung et al.

Array or memory accesses in which the index/address is symbolic can also lead to path explosion depending on their implementation. A simple approach is for a symbolic execution engine to choose a single concrete value for the address which is consistent with the path condition, and then to perform the operation with that address. This leads to relatively simple constraints on any particular path, but if many different address values were possible, the symbolic execution engine would need to cover all of them on other execution paths to cover the whole behavior of the code. The table accesses used in applying a table-based CRC implementation to symbolic data are examples of access that would cause path explosion for this reason.

This has motivated a variety of other techniques for symbolic execution tools to deal with symbolic array indexes and memory accesses. In a full system it is desirable to have techniques to cover stores as well as loads, and in a binary-level tool it is not trivial to determine the bounds of an array. But here we consider just the case relevant to CRC implementations, of loads from a memory area with a known range.

If one wishes to avoid branching when a symbolic execution tool reaches a load from an array with a symbolic address, the tool must construct a single symbolic expression which describes the values that might be loaded for any feasible value of the address. Sticking within the bit-vector formulas used for other machine operations, one natural approach (especially if the array size is a power of two) is to build a balanced tree of if-then-else operations. This tree has leaf nodes that correspond to values in the table and internal nodes that correspond to check each bit of the index expression. We present an example in Listing 2.

```
1 t1 = (cast(input0_0:reg8_t)S:reg32_t & 3:reg32_t)
2 t2 = cast(t1:reg32_t)L:reg1_t
3 t3 = cast(
4        (cast(input0_0:reg8_t)S:reg32_t & 3:reg32_t)
5          >> 1:reg8_t
6        )L:reg1_t
7 t4 = t3:reg1_t ? (t2:reg1_t ? 3:reg32_t : 2:reg32_t)
8                : (t2:reg1_t ? 1:reg32_t : 0:reg32_t)
```

Listing 2. Table treatment formula produced by FuzzBALL

Listing 2 shows a formula generated for loading a 32-bit value from a table that contains four values, 0, 1, 2, and 3. Line 1 produces an index expression that ranges between 0 and 3. Lines 2-6 produces a predicate expression that check the two least significant bits of the index expression. Finally, lines 7,8 load the appropriate value from the table based on each of two least significant bits of the index expression.

Instead of coming up with clever ways to encode the contents of the array in which a lookup occurs, a symbolic executor can instead choose to provide the values in the array to the solver in the query. The symbolic executor can also create a clause in the query that chooses a value from the array using a variable. This method passes the complexity of reasoning about the resulting value to the solver. Many solvers such as Yices [14], Boolector [7], and Z3 [11] have support for the theory of arrays. (In solver terminology, the theory of arrays also includes efficient representations of functional updates to arrays, which are useful for encoding stores. Read-only tables can also be encoded using a related feature called “uninterpreted functions”.)

Another opportunity for optimization is that if the contents of a lookup table have a systematic form, it may be more efficient to express the formula behind the table directly, instead of listing the values. In the if-then-else tree representation described above, large regions of the table with a constant value can be simplified in the formula. Cha et al.’s Mayhem system [9] created simple summaries for regions of tables where the values increased linearly, such as the mapping from uppercase to lowercase letters. The lookup tables used in CRC computations are not constant or linear in the sense of these previous optimizations, but they have another structure that can be recognized to encode them efficiently.

We say that a lookup table T is “GF(2)-linear” if, for indices i and j for which i, j, and $i \oplus j$ are within the table bounds, $T[i \oplus j] = T[i] \oplus T[j]$, where as usual $\oplus$ represents bitwise XOR. The lookup tables used to implement CRCs generally inherit a GF(2)-linear structure from the CRC operation itself. An efficient formula representing a GF(2)-linear table, and an efficient algorithm for detecting such a table, come from observing that all the values in the table are determined by the values at locations whose index has exactly one bit set. We call these values the “spine” of the table. Every entry in the table can be represented as the XOR of the spine values for each of the bits set in its index; or equivalently, as the XOR of values that are each a spine value or 0 based on whether the corresponding bit in the index is set. A table is GF(2)-linear if every value is equal to the one that would be computed based on the formula from its spine.

V. Evaluation

A. Setup

We obtained two implementations of CRC32 computation and one implementation of CRC64. Our first CRC32 implementation was obtained from a reference provided by Jung et al. [18]. This implementation is not optimized for speed and therefore does not use a lookup table. It instead computes the values in the table dynamically using branching instructions. Our second CRC32 implementation came from the EternalPass challenge which is part of the DARPA CGC benchmark suite [10]. This implementation uses values from a lookup table containing 256 entries to compute the CRC. Finally, we obtained an implementation of CRC64 from the liblzma compression library of XZ Utils [12]. This CRC64 implementation also uses a lookup table. We combined every CRC implementation with a test harness shown in Listing 3. The test harness attempts to use a symbolic execution and a solver to construct input values whose CRC matches the CRC of some random concrete byte values. This test harness has the same structure as the anti-fuzzing change made by Jung et al.'s tool to impede symbolic execution.

On line 1, the test harness puts `LEN` symbolic bytes into `sym_str`. Next, it runs the CRC implementation on `sym_str` and saves its result in `sym_crc` on line 6. On lines 9-11, the test harness writes `LEN` concrete bytes into `conc_str`. Finally, the test harness checks whether the CRC computed from `LEN` symbolic bytes in `sym_str` matches the CRC computed from `LEN` bytes with random values on line 13.

```
1  char sym_str[LEN]; // contains LEN symbolic bytes
2  void test_harness() {}
3    // initializes lookup table, if any
4    crc_init();
5    // compute CRC of symbolic bytes
6    uint64_t sym_crc = crc(sym_str, LEN, INITIAL_CRC);
7    char conc_str[LEN]; // LEN = no. of input bytes
8    // initialize LEN concrete bytes to random values
9    srand(time(NULL));
10   for (int i = 0; i < LEN; i++)
11     conc_str[i] = (char) rand();
12   // compare the two CRCs
13   if (sym_crc == crc(conc_str, LEN, INITIAL_CRC)) {
14     printf("found a bug\n");
15   }
16 }
```

Listing 3. Test harness to run CRC with symbolic inputs

Next, we ran Jung et al.'s CRC32 implementation, which does not use a lookup table, with two path-merging symbolic executors, angr [26] and Java Ranger [23]. angr can be configured to turn on a path-merging technique named veritesting [3]. Java Ranger is an open-source symbolic executor [24] that extends path-merging for Java bytecode by statically summarizing multi-path code regions that contain method calls. Java Ranger is built as an extension to the Symbolic PathFinder tool [21]. We ported the Jung et al.'s CRC32 implementation to Java to run it with Java Ranger.

We ran the test harness with the CRC32 and CRC64 implementations that use a lookup table with FuzzBALL [20]. During this part of the evaluation, we turned on one of three different optimizations that FuzzBALL uses for lookups in an array. These three optimizations are used when FuzzBALL runs into a lookup into a memory region (such as an array) with a concrete base address using a symbolic index expression. These three features avoid the path explosion resulting from the symbolic executor branching once for every possible index value.

1) The first optimization, which we call "ITE tree table", produces a balanced tree of if-then-else formulas. The leaves of this tree are the values in the array and the internal nodes represent a if-condition that checks each bit of the symbolic index expression. Typically, this optimization tends to construct large formulas since every value present in the array has to be present in this balanced tree of if-then-else formulas too. This feature served as a baseline for our comparison.
2) The second optimization, which we call "GF(2)-linear table", checks if all the values in the array can be computed using an XOR function. If this check succeeds, this optimization computes the result of the lookup as an XOR function.
3) The third optimization enables use of the theory of arrays support from the solver that FuzzBALL is configured to use. We use "theory of arrays table" to refer to this feature. This optimization allows FuzzBALL to communicate to the solver the values in the lookup table. This optimization ultimately relies on optimizations in the solver for performance.

Finally, for all the four symbolic executor optimizations, path-merging, ITE tree table, GF(2)-linear table model, and use of theory of arrays solver support, we compared the performance of three solvers, Yices [13], Boolector [7], and Z3 [11].

For every run of the test harness with a symbolic executor, we gave it an input of 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, or 8192 unconstrained symbolic bytes. Each run was allowed a two hour time budget. We ran every number of unconstrained symbolic bytes input 10 times with the same number of concrete input bytes set to random values, to average out variation caused by the randomized search finding a solution earlier or later by chance. All the runs were done on a machine running Ubuntu 16.04.6 LTS with 16 Intel(R) Xeon(R) CPU E5-2623 v3 @ 3.00GHz CPU cores with a total of 192 GB RAM. Each individual symbolic execution and solving task was single-threaded. The evaluation with angr and Java Ranger were given a memory limit of 8 GB. The next two subsections will respectively describe the results of each symbolic executor optimization and compare these results with performance across different solvers.

B. Results

We present the result of comparing FuzzBALL's three table lookup optimizations for the CRC32 and CRC64 implementations in Figures 1 and 2 respectively. Both Figures 1 and 2 report average running time of the symbolic executor over 10 runs with different random concrete values. Both figures show that using ITE tree table is the slowest in finding an input whose CRC equals the CRC of random values. Both Figures 1 and 2 show that GF(2)-linear table lookup is the fastest feature for inputs up to 16 symbolic bytes. Figure 1 shows that the GF(2)-linear table lookup is slower than the theory of arrays table lookup for up to 512 symbolic inputs. However, at 1024 symbolic inputs, these two features are equally fast at finding an input 1024 bytes long whose CRC equals the CRC of 1024 random bytes. For symbolic inputs longer than 1024 bytes, the GF(2)-linear table lookup is again the fastest feature. Similarly, Figure 2 shows a similar trade-off happens at 512 symbolic inputs. For symbolic inputs longer than 512 bytes, the GF(2)-linear table lookup is the fastest among the three techniques at finding a correct sequence of byte values.

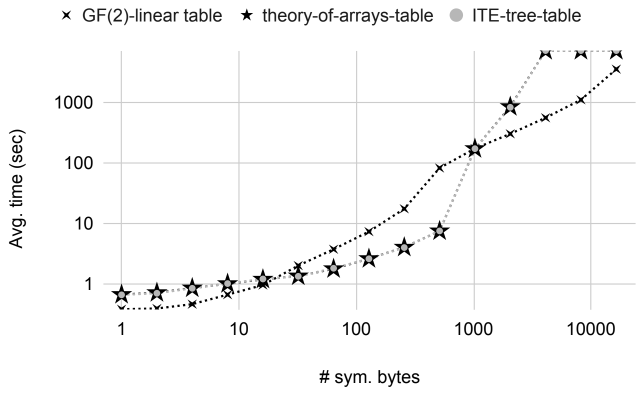

Fig. 1. Comparing three independent methods of executing a lookup into a CRC32 lookup table

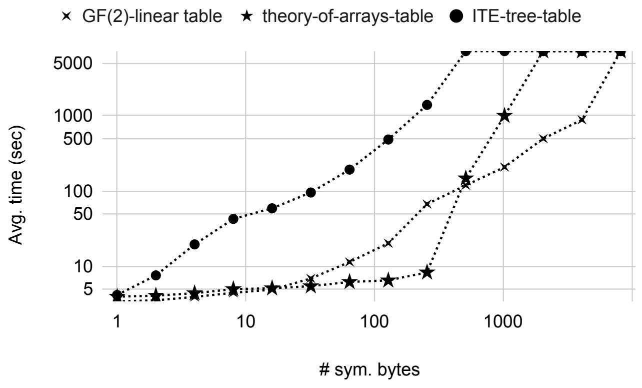

Fig. 2. Comparing three independent methods of executing a lookup into a CRC64 lookup table

CRC implementations that are optimized for speed often use lookup tables. The CRC implementation provided by Jung et al. was instead optimized to lead a symbolic executor into path explosion with branching for each bit of the input. Since path-merging is a technique that attempts to overcome such kinds of path explosion, we ran Jung et al.'s CRC32 implementation using angr and Java Ranger, two symbolic executors which have the path-merging feature.

Running Jung et al.'s CRC32 under angr confirms how it hinders the symbolic execution. Figure 3 shows the average time required by angr with and without path-merging to find the correct value for the magic number. angr fails to find magic number for any symbolic input of more than 2 bytes. We tried the `spiller` exploration technique to avoid memory problems but still angr was not able to finish the task within the 2 hour time limit. When path-merging was enabled, angr was able to find the magic number for up to 30 symbolic input bytes. This shows how path-merging helps with symbolic execution of non-table-based CRC implementation.

angr also has a feature to support generating path constraints from an entire lookup table [6] which we investigated, but we were not able to set up an appropriate experimental comparison as of this submission. In the example we tried, it appeared that angr did successfully generate a lookup table but it concluded that the path, on which the CRC of the symbolic input matched the concrete target, was infeasible. We have not yet determined the cause of this failure.

However, we use the memory model of angr as described by Borzacchiello et al. to make a few observations. The CRC lookup tables used in our evaluation consisted of 256 4-byte entries. Each read from a CRC lookup table would use a symbolic pointer expression because of the index being symbolic. On encountering such reads, as per Borzacchiello et al., angr would conclude that this symbolic read can span at most 1024 bytes. Since the number of bytes that this symbolic read can span is equal to angr's default threshold of 1024 bytes, this would cause angr to create an unbalanced tree of if-then-else expressions with all possible results of the read captured at the leaves of the unbalanced tree. This generated if-then-else expression closely matches the expression generated by FuzzBALL's ITE tree table feature. Therefore, we expect the performance of FuzzBALL's ITE tree table feature to closely match angr's behavior on CRC table-based implementations.

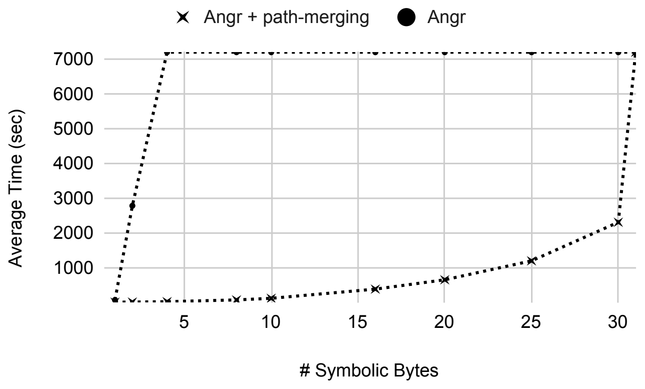

Fig. 3. Running time for exploring Jung et al.'s CRC32 implementation under angr with and without path-merging

Java Ranger's path-merging is an extension of veritesting for Java bytecode. We present the results of running Jung et al.'s CRC32 implementation with Java Ranger in Figure 4. We found that the baseline symbolic executor, SPF, times out in 2 hours when exploring this CRC32 implementation with more than a single byte of input. Java Ranger's path-merging, on the other hand, can allow symbolic execution to scale up to 8192 symbolic input bytes. We also attempted running a lookup table-based CRC32 implementation with SPF and Java Ranger. On encountering a lookup into a table with 256 entries using a symbolic index, both tools create 256+2 branches to symbolically execute the array lookup. One branch is created for exploring the symbolic index being equal to the index for each of the 256 entries. Two other branches are created for exploring the symbolic index being out-of-bounds in the lookup table. Because this approach has to explore 258^N execution paths for a N-byte symbolic input, it clearly does not scale well. We found that this approach timed out with even 2 symbolic input bytes in a 2 hour time budget. Given that table-based CRC implementations don't contain any multi-path code regions that would be statically summarized by a path-merging symbolic executor, we have excluded running path-merging symbolic executors, angr and Java Ranger, on table-based CRC32 implementations.

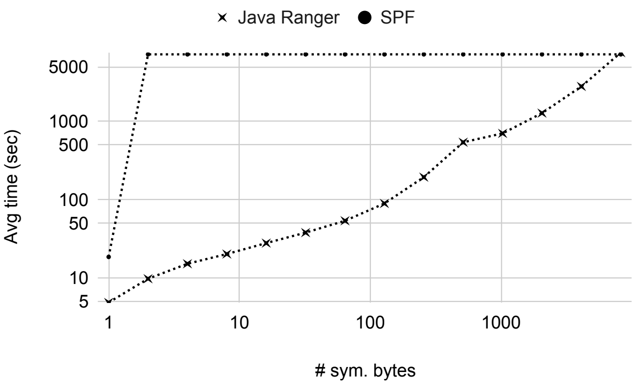

Fig. 4. Running time for exploring Jung et al.'s CRC32 implementation with Java Ranger

Apart from the four optimizations that may be performed by the symbolic executor, the choice of solver can also influence the total time it takes to find an input whose CRC matches the CRC of a given value. Therefore, we saved the query that asks the solver to find such inputs into a file with the SMT2 file format. Next, we provided this query file to Yices2 and Boolector to see if they are faster than Z3 at solving these queries. We saved such query files for every length of symbolic input ranging from 1 to 8192 symbolic inputs. To get such query files for the range of input sizes, we ran Java Ranger on Jung et al.'s CRC32 implementation and FuzzBALL on liblzma's CRC32 implementation. We present the results of this comparison with Yices2 and Boolector in Figures 5 and 6 respectively. Figures 5 and 6 show that the GF(2)-linear table lookup feature produces queries that are fastest to solve across Yices2 and Boolector. Both solvers do not perform well on theory of arrays table lookup. Using ITE tree table performs better than path-merging starting from 2048 symbolic input bytes in Boolector. On the contrary, path-merging performs better than ITE tree table starting from 2048 symbolic input bytes in Yices2.

Table I shows that which solver performs best with the four optimization techniques. Both Yices2 and Boolector are much faster than Z3 at solving queries generated by FuzzBALL's GF(2)-linear table lookup feature. Overall, Yices2 shows the best performances on GF(2)-linear table lookup, ITE tree table and path-merging.

VI. Discussion

When searching for CRC implementations to use for comparing optimizations in symbolic executors, we found two kinds, ones which used a lookup table and others that didn't. We found that implementations that used lookup tables were optimized for performance. For example, the CRC32 implementation in Apache Hadoop [1] claims to be 1.8 to 10 times faster than the CRC32 implementation in the Java standard library that executes CRC computation using native execution. To symbolically explore such CRC implementations, we have so far found the GF(2)-linear table optimization to be fastest. We also found some CRC table-based implementations that were minor variations of the ones used in our evaluation. For example, the Apache Hadoop CRC implementation combines 8 tables with 256 entries each into a single table with 2048 entries. To address such variations, we re-implemented our GF(2)-linear table lookup in Java Ranger with a minor tweak that detects the base address of the table. This change still allowed Java Ranger to scale up from 1 symbolic input byte

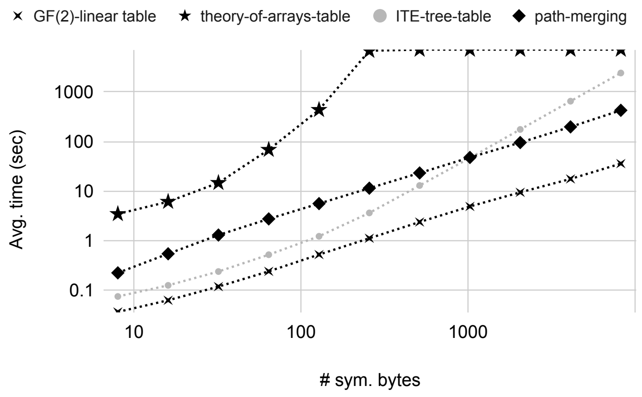

Fig. 5. Performance comparison of four methods with table-based CRC32 implementation using Boolector

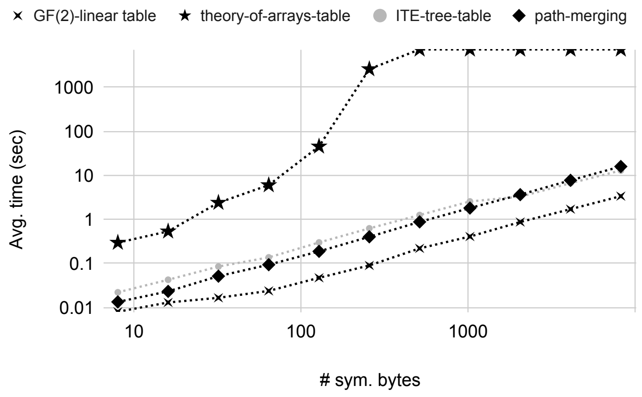

Fig. 6. Performance comparison of four methods with table-based CRC32 implementations using Yices2

TABLE I. Performance comparison of four methods for CRC32 implementations with 8192 symbolic input bytes using different solvers (2-hour time limit)

	Boolector	**Yices2**	**Z3**
	Ver. 3.1.0	Ver. 2.6.1	Ver. 4.4.0
GF(2)-linear-table	35.801 s	3.366 s	1028.781 s
theory-of-arrays-table	TO	TO	TO
ITE-tree-table	2426.298 s	12.852 s	TO
path-merging	427.375 s	15.841 s	3951.699 s

to 32 symbolic inputs in a 2 hour time budget.

Other CRC implementations, such as the one provided by Jung et al., may instead choose to be optimized for simplicity. To avoid path explosion in such implementations, path-merging support in symbolic executors such as angr needs to be optimized for performance.

Though we have not applied these tools in an end-to-end experiment against the AntiHybrid obfuscation method proposed by Jung et al., these results suggest that such obfuscation is not sufficiently challenging to be relied upon to frustrate symbolic execution. Our results show that the constraints arising from a 4 or 8-byte obfuscated comparison can be solved in less than a 0.1 seconds using existing symbolic execution and SMT solver combinations. This is still longer than the time needed to resolve a simple non-obfuscated equality comparison, which would be near instantaneous if there are no conflicting constraints. However these results suggest that CRC-derived constraints are not substantially more difficult to resolve than some other constraints that arise commonly in symbolic execution. If building a symbolic execution system specifically resistant to Jung et al.'s obfuscation, it would also be valuable to ensure that the effort involved in inverting each CRC-obfuscated comparison was cached.

VII. Conclusion

Given that CRCs have been found to not be resistant to adversarial modification in other contexts such a cryptography, we argue that this lack of difficulty also carries over to symbolic execution of CRCs. We found two implementation variants of CRC being used in the wild and presented techniques to overcome difficulties posed by their symbolic execution. We found that path-merging easily overcomes the path explosion barrier posed by CRC implementations that avoid use per-bit branching. For CRC implementations that use a lookup table, we compared three different table lookup optimizations in FuzzBALL. We found that FuzzBALL's GF(2)-linear table lookup optimization is the fastest at solving the pre-image computation constraint across three solvers, Z3, Yices2, and Boolector. Using a combination of path-merging and GF(2)-linear table lookup overcomes any difficulty of computing a pre-image for a symbolic executor. Given our results, we argue that CRCs should no longer be considered a building block for constructing barriers that impede symbolic execution.

Acknowledgment

The research described in this paper has been supported in part by the National Science Foundation under grants 1563920 and 1815621, and by the Office of Naval Research under grant N00014-19-1-2541.

References

[1] Apache Software Foundation, "PureJavaCRC32C Apache Hadoop Main 2.7.4 API," Dec. 2019. [Online]. Available: https://hadoop.apache.org/docs/r2.7.4/api/org/apache/hadoop/util/PureJavaCrc32C.html

[2] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: fuzzing with input-to-state correspondence," in *26th Annual Network and Distributed System Security Symposium, NDSS*, Feb. 2019.

[3] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 1083–1094. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568293

[4] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, no. 3, 2018.

[5] N. Borisov, I. Goldberg, and D. A. Wagner, "Intercepting mobile communications: the insecurity of 802.11," in *MOBICOM 2001, Proceedings of the Seventh Annual International Conference on Mobile Computing and Networking*, Jul. 2001, pp. 180–189.

[6] L. Borzacchiello, E. Coppa, D. Cono D'Elia, and C. Demetrescu, "Memory models in symbolic execution: key ideas and new thoughts," *Software Testing, Verification and Reliability*, vol. 29, no. 8, p. e1722, 2019, e1722 stvr.1722. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1722

[7] R. Brummayer and A. Biere, ""Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays"," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Kowalewski and A. Philippou, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 174–177.

[8] J. Caballero, P. Poosankam, S. McCamant, D. Babic, and D. Song, "Input generation via decomposition and re-stitching: Finding bugs in malware," in *Proceedings of the 17th ACM Conference on Computer and Communication Security (CCS)*, Chicago, IL, October 2010.

[9] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing Mayhem on binary code," in *IEEE Symposium on Security and Privacy*, May 2012, pp. 380–394.

[10] DARPA, "DARPA Cyber Grand Challenge," Jun. 2018. [Online]. Available: https://github.com/CyberGrandChallenge/

[11] L. de Moura and N. Bjørner, ""Z3: An Efficient SMT Solver"," in *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and J. Rehof, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340.

[12] T. Developers, "XZ Utils," Dec. 2019. [Online]. Available: https://tukaani.org/xz/

[13] B. Dutertre, "Yices 2.2," in *Computer Aided Verification - 26th International Conference CAV*, Jul. 2014, pp. 737–744.

[14] B. Dutertre and L. De Moura, "The YICES SMT solver," *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, vol. 2, no. 2, pp. 1–2, 2006.

[15] A. Futoransky and E. Kargieman, "CORE-SDI-04: SSH insertion attack," Jun. 1998. [Online]. Available: https://marc.info/?l=bugtraq&m=93656900402840

[16] P. Godefroid, "Higher-order test generation," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 258–269. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993529

[17] T. Hansen, P. Schachte, and H. Søndergaard, "State joining and splitting for the symbolic execution of binaries," in *Runtime Verification, 9th International Workshop, RV 2009, Grenoble, France, June 26-28, 2009. Selected Papers*, 2009, pp. 76–92.

[18] J. Jung, H. Hu, D. Solodukhin, D. Pagan, K. H. Lee, and T. Kim, "Fuzzification: Anti-fuzzing techniques," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1913–1930. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/jung

[19] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 193–204.

[20] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration Lifting: Hi-fi Tests for Lo-fi Emulators," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII. New York, NY, USA: ACM, 2012, pp. 337–348. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151012

[21] C. S. Păsăreanu and N. Rungta, "Symbolic PathFinder: Symbolic Execution of Java Bytecode," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ser.

ASE '10. New York, NY, USA: ACM, 2010, pp. 179–180. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859035

[22] K. Sen, G. Necula, L. Gong, and W. Choi, "Multise: Multi-path symbolic execution using value summaries," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 842–853. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786830

[23] V. Sharma, S. Hussein, M. Whalen, S. McCamant, and W. Visser, "Java Ranger at SV-COMP 2020 (competition contribution)," in *Proc. TACAS (2)*, ser. LNCS 12079. Springer, 2020.

[24] V. Sharma, S. Hussein, M. W. Whalen, S. McCamant, and W. Visser, "Java Ranger," Dec. 2019. [Online]. Available: https://github.com/vaibhavbsharma/java-ranger

[25] V. Sharma, M. W. Whalen, S. McCamant, and W. Visser, "Veritesting challenges in symbolic execution of Java," *SIGSOFT Softw. Eng. Notes*, vol. 42, no. 4, pp. 1–5, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3149485.3149491

[26] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice: Automatic Detection Of Authentication Bypass Vulnerabilities In Binary Firmware," in *NDSS*, 2015.

[27] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution," in *23rd Annual Network and Distributed System Security Symposium, NDSS*, Feb. 2016.

[28] T. Wang, T. Wei, G. Gu, and W. Zou, "Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *31st IEEE Symposium on Security and Privacy, S&P 2010*, May 2010, pp. 497–512.

[29] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM: A practical concolic execution engine tailored for hybrid fuzzing," in *27th USENIX Security Symposium*, Aug. 2018, pp. 745–761.