

# Automatic Formal Verification of Block Cipher Implementations

Eric Whitman Smith and David L. Dill  
Gates Bldg. 3A, Stanford University, Stanford, CA 94305  
{ewsmith,dill}@cs.stanford.edu

*Abstract*—This paper describes an automatic method for proving equivalence of implementations of block ciphers (and similar cryptographic algorithms). The method can compare two object code implementations or compare object code to a formal, mathematical specification. In either case it proves that the computations being compared are bit-for-bit equivalent.

The method has two steps. First the computations are represented as large mathematical terms. Then the two terms are proved equivalent using a phased approach that includes domain-specific optimizations for block ciphers and relies on a careful choice of both word-level and bit-level simplifications. The verification also relies on STP [5], a SAT-based decision procedure for bit-vectors and arrays. The method has been applied to verify real, widely-used Java code from Sun Microsystems and the open source Bouncy Castle project. It has been applied to implementations of the block ciphers AES, DES, Triple DES (3DES), Blowfish, RC2, RC6, and Skipjack as well as applications of the cryptographic hash functions SHA-1 and MD5 on fixed-length messages.

## I. INTRODUCTION

A block cipher is a cryptographic algorithm that encrypts a small block of data using a shared, secret key. Block ciphers are critical infrastructure underlying important national security and e-commerce applications. Therefore, their correctness is crucial, and formal verification of block cipher implementations is worthwhile, especially if it can be accomplished with reasonable effort.

The verification method described in this paper can prove mathematically that two computations are equivalent. For block ciphers this means that they produce identical outputs for every possible message and key. To verify an implementation, one can prove it equivalent to either a reference implementation that is strongly believed to be correct, or to a carefully written formal mathematical specification. Such a specification is typically written independently of any implementation and usually very closely resembles the official informal specification of the cipher. Even if there is no formal specification or golden model, proving the equivalence of two implementations produced independently can increase one's confidence in both of them. Two implementations proven equivalent are either both correct, or they both have the same bugs.

A key property of block cipher code is that the loops can be completely unrolled. That is, they have static upper limits on the number of times that they may execute. Usually this is because the cipher consists of a fixed number of “rounds” (e.g., 10 rounds for AES-128 encryption). The verification method unrolls all loops and recursion in the implementations

and specifications of the ciphers, generating large loop-free terms. Unrolling avoids the need to specify loop invariants or verify the loops using inductive methods. It remains to prove equivalence of the resulting large terms, and the automation of that process is the focus of this paper.

This paper makes several new research contributions. Most importantly, it demonstrates the feasibility of automatically formally verifying existing block cipher implementations written in a real, widely-used language. It also identifies the key issues in performing such verifications and explains how they can be addressed.

## II. RESULTS

Our method has been applied to verify many real-world block cipher implementations written in Java, including all of the block ciphers in Sun's implementation of the Java Cryptography Extension (AES, DES, Triple DES, Blowfish, and RC2) and several ciphers from the open source Bouncy Castle project (three AES implementations, Blowfish, DES, Triple DES, RC2, RC6, and Skipjack). All implementations have been shown to be bit-for-bit equivalent, for all messages and all keys of the given length, to the corresponding formal mathematical specifications. Also, several proofs were done equating Sun and Bouncy Castle implementations of the same ciphers directly, to show that implementations can be compared without the use of formal specifications.

For the AES cipher, four implementations have been verified (three from Bouncy Castle and one from Sun). For each implementation both the encryption and decryption operations were verified for all three official key lengths. The implementations are heavily optimized. Even the simplest one (AESLightEngine) packs the sixteen bytes of the cipher state into four 32-bit machine words and partially unrolls the loops to perform two rounds of AES per loop iteration. AESEngine, AESFastEngine and Sun's AESCrypt are further optimized with lookups into large pre-computed tables. For the ciphers that support many key lengths (RC2 and Blowfish), we have verified only one key length (but other key lengths should be similar).

We also verified implementations of the cryptographic hash functions MD5 and SHA-1 from Bouncy Castle. Unlike block ciphers, these functions take input messages of unknown length. Thus, the method does not directly apply, because the number of input bits is not fixed and the loops cannot be completely unrolled. However, one can fix the length of the

input messages and then apply the method. We constrained the MD5 and SHA-1 implementations to each run on 32-bit and 512-bit messages, and then proved them equivalent to the formal specifications.

The proof of a typical cipher can be rechecked by a computer in a matter of minutes or a few hours. For AES-128 encryption, unrolling the specification takes about 7 seconds on a fast machine. Symbolically executing the Java code for the Bouncy Castle’s “light,” “regular,” and “fast” implementations takes 14, 10, and 16 seconds, respectively. The equivalence proofs connecting the unrolled implementations to the unrolled formal specification take 51, 178, and 167 seconds respectively.

We found that we can often verify a new cipher implementation in a day or two. Our most recent cipher verification, for Skipjack, took less than three hours, including time spent writing and debugging the formal specification.

### III. BACKGROUND AND RELATED WORK

The standard approach to validating block cipher implementations is testing. For example, putative AES implementations must pass a suite of tests from the National Institute of Standards and Technology [9]. However, the number of possible inputs to a block cipher is too large to test exhaustively; there are  $2^{256}$  possible inputs for 128-bit AES, which takes a 128-bit message and a 128-bit key. Indeed, any block cipher that did allow for exhaustive testing would likely be insecure, since one could recover a plaintext by trying all possible decryption keys.

There have been other efforts to apply formal methods to block ciphers. For example, Duan, Hurd, Li, Owens, Slind, and Zhang used an interactive theorem prover to prove that decryption inverts encryption for several block ciphers expressed in higher order logic [4]. Their approach seems to require significant manual effort to guide the theorem prover. Also, the inversion property, while important, is fairly weak; insecure implementations that do not conform to the AES standard may still satisfy inversion (consider the trivial implementation that does nothing at all for both encryption and decryption). By contrast, our verification method proves bit-for-bit equivalence between each implementation and a formal specification or a reference implementation. Also, our approach applies to real-world implementations in Java bytecode (rather than the native language of a theorem prover).

Toma and Borrione used the ACL2 interactive theorem prover to verify a hardware implementation of SHA-1 [12]. Their work also seems to involve significant manual effort to guide the prover.

Galois Connections is developing Cryptol, a domain specific language for cryptography which can be compiled down to an implementation using a verifying compiler [3] [10]. (The same seems possible for the specifications of block ciphers analyzed by Duan et. al.) In contrast, our work verifies pre-existing block cipher implementations not written using a correct-by-construction framework.

Yin, Knight, Nguyen, and Weimer, have applied their Echo tool to verify an implementation of AES [14] by repeatedly refactoring the code to remove optimizations. Their approach seems less automatic than ours because the user must often identify the transformations to be performed (e.g., finding instances of word packing or specifying the patterns encoded in the values of lookup tables). Our work makes no such requirements.

Sean Weaver, in a presentation posted online, has described a verification method similar to the equivalence checking phase of our method [13]. Both methods decompose large equivalence proofs by identifying pairs of probably equal nodes using random test cases, and both make calls to boolean satisfiability solvers. Weaver has applied his method to verify an AES implementation, but his presentation does not address other ciphers. We found AES to be among the easiest ciphers to verify, and many of the complications we address later in this paper arose in the verification of other ciphers.

Our approach borrows several techniques from combinational equivalence checking, which is commonly used in hardware verification. In particular, the use of random test cases to find internal equivalences goes back at least to Berman and Trevillyan’s work in 1989 [2]. Their approach, as does Kuehlmann’s [6], involves proving the discovered equivalences in a bottom-up fashion – much like our method. Early approaches used binary decision diagrams, but state-of-the-art work on combinational equivalence checking, including Kuehlmann [15]’s later work and the checker ABC [7], involves “SAT-sweeping” or “fraiging,” in which functionally equivalent nodes in a miter are identified through simulation and proved equivalent by a satisfiability solver. Our equivalence checking phase is very similar.

A key distinguishing feature of our approach is that it uses word-level simplifications to normalize terms as much as possible. (By “word-level” we mean simplifications performed before “bit-blasting,” when the terms involved are still entire bit-vectors rather than single bits.) Our simplifications are not guaranteed to give identical representations for equivalent terms, but our approach proves to work well in practice; simplifying transformations alone suffice to verify many block cipher implementations, with no equivalence checking phase required.

### IV. THE VERIFICATION METHOD

#### A. Inputs to the method

The verification method takes as input a collection of compiled Java class files that together implement a cryptographic algorithm (e.g., the cipher engine class, its helper classes, and its ancestor classes and interfaces). Verifying class files instead of source code has several advantages. First, class files contain the actual bytecodes that will run on the machine, and object code verification can thus catch errors introduced during compilation. Second, source code may not always be available for analysis. Finally, it is convenient to model Java bytecode using operational semantics, using an executable formal model of the Java Virtual Machine. (The model is

derived from the M5 model of Moore et al. [8].) Unlike many formal verification frameworks, no programmer annotation of the source code is required.

Our method can compare two Java implementations of the same algorithm, but often a Java implementation is instead compared to a formal, mathematical specification. Such a specification must be written by hand by translating the document that defines the cipher (e.g., FIPS-197 for AES [1]) into a formal language. Writing and debugging a specification for a new cipher typically takes a few hours. Once the specification is written it can be reused to verify many implementations of the algorithm. Formal specifications have been written for AES, DES, Triple DES, Blowfish, RC2, RC6, Skipjack, MD5, and SHA-1. Each is written in the native language of the ACL2 theorem prover, a side-effect-free dialect of Common Lisp with well-defined semantics. The specifications are written without optimizations, so as to be as close to the official documents as possible. They are also executable and can be validated using test cases. For example, we have tested our AES specification by running it on all the test cases given in the standard.

The user of our verification method must provide a few simple inputs. First, the user must write a driver that calls the cryptographic algorithm to be verified. The driver is a small Java method that takes an input message and a key and calls the cipher in the typical way. Usually this involves allocating a new object of the cipher class, perhaps wrapping its key in a “key parameter” object (for the Bouncy Castle ciphers), perhaps calling an initialization routine, then calling the main encryption routine, and finally perhaps calling a finalization routine (for the cryptographic hash functions). Such a driver would be easy to write for anyone capable of writing code that uses the cipher. Also, drivers for different ciphers implementing the same Java interface or abstract class (e.g., `org.bouncycastle.crypto.BlockCipher`) are often very similar.

The user must also specify the Java class files containing the code to be executed. (We could improve our tool to generate this list automatically.)

Finally, the user must list the input variables and their sizes and indicate how the inputs of the two computations match up (as must always be done when making a miter). Handy macros are provided to help. Given these inputs, the method runs automatically.

### *B. Terms and simplifications*

The verification method has two steps. First, the two computations to be compared are represented as large mathematical terms including bit-vector and array operations. Then the terms are compared to establish that they compute identical outputs for all possible inputs. The comparison is the main focus of this paper, but first we briefly describe how terms are represented and simplified, and how suitable terms are generated from Java bytecode and formal specifications.

Computations are represented as large mathematical terms. Identical subterms are shared, so the terms are really directed acyclic graphs. The “nodes” of a term include its input

variables (which together determine its value), constant nodes, and nodes which represent function applications. A function application includes an operator and a list of arguments (constants and other nodes called the node’s “children”). Nodes are numbered, and we require that a node have a higher number than any of children. Thus our terms are acyclic. Our terms differ from operator trees in an important way: shared subterms are represented only once (that is, we perform “structural hashing”). This is crucial because block cipher computations perform extensive mixing operations in which results from previous rounds appear many times in the expressions for future rounds. Sharing subterms results in exponential space savings when representing these computations. Our representation of terms is also generic. The operator of a node can be any function in the ACL2 logic, such as a recursive Lisp function from a formal specification or one of the functions used to model the complicated state of the Java Virtual Machine. Through unrolling and simplification, terms containing only bit-vector and array operators can be obtained.

The loop-free terms representing block ciphers are large. For example the term for Sun’s Blowfish encryption operation using a 64-bit key has 220,811 nodes after simplification (and before word operations are bit-blasted into bit operations).

It is often important to simplify or transform terms, and a novel tool is employed to do so by applying simplification rules. The rules can range from simple identities to sophisticated transformations that handle common coding idioms. Each simplification replaces a subterm with an equivalent but simpler term, and successful simplifications often enable further simplifications later on. The careful choice of which simplifications to perform, and the order in which to perform them, is important to successful verification. Simplifications serve several purposes. First, they reduce the size of terms involved in the verification effort. More importantly, they often allow for the normalization of terms; that is, logically equivalent terms are transformed to have the same syntactic form (and so are clearly equal by inspection). We have found normalization to be crucial to the verification effort. In many cases, normalizing terms using our rules is sufficient in itself to prove the equivalence of two cipher implementations. This is the case for the RC2, RC6, Blowfish, Skipjack, and “light” AES implementations from Bouncy Castle and for the RC2 and Blowfish implementations from Sun. For these examples, two completely independent representations of the same computation (one in Java bytecode and the other in ACL2’s language of recursive functions) can be proved equivalent just by applying simplifications. The tool used to simplify terms is similar to the simplifier of the ACL2 theorem prover and borrows several key ideas from it. However, a crucial difference is that our tool can efficiently handle terms with many shared subterms. ACL2’s inability to do so makes it unable to handle the unrolling of block ciphers. The simplification rules applied by the term simplifier are stated as ACL2 theorems and can be proved. (We have proved most of our rules and are adding proofs for the rest.)

### C. From bytecode and specifications to mathematical terms

In order to verify a Java bytecode implementation of a block cipher, one must first extract a term representing the core computation of the cipher. Our tool for doing so uses symbolic execution and the term simplifier described above. While not the main focus of this paper, we briefly describe the process here.

Extracting the computation performed by a bytecode program is a non-trivial task. The Java bytecode language includes many complicated concepts, including field and method resolution, allocation of new heap addresses, static initializers of classes, the use of values from the runtime constant pool, string interning, the throwing of exceptions, etc. Surprisingly, many of these complicated language features are used incidentally in common block cipher implementations. To extract the core computation performed by a program, the tool symbolically simulates the operation of the Java Virtual Machine (JVM). The simulation is done by using the term simplifier to repeatedly simply an application of the “run until return” function when the machine is executing the driver method. The details are beyond the scope of this paper, but the simulation essentially steps through the computation one instruction at a time until the driver method finally returns. This amounts to a full unrolling of all the loops and a full inlining of all subroutine calls. Our approach is very similar to that used by J Moore et. al. with their ACL2 models of the JVM. Key differences are that we use an improved version of Moore’s M5 machine model which is more amenable to formal reasoning and that our terms share common subterms. Our tool also handles conditional branches in a sophisticated way to prevent some simulations from splitting into exponentially many cases. (Briefly, it simulates the branches independently until flow reconverges and then combines the resulting states, using an if-then-else operator for any state components that differ between the branches.)

An important feature of our symbolic simulation is that it simplifies terms as it goes. This keeps the term size manageable and can prevent the simulations from encountering exceptions or error states. For example, the JVM performs a bounds check for each array access and throws an exception if the check fails. The on-the-fly application of simplifications allows such bounds checks to be discharged during the simulations. The symbolic simulation process can extract bit-accurate representations of the computations performed by long symbolic executions (tens of thousands of Java bytecodes, with tens of thousands of nodes in the resulting mathematical terms).

Formal specifications must be unrolled into terms involving only bit-vector and array operations. The term simplifier is used to expand and unroll all other functions, including recursive functions. At each stage of the unrolling, it is usually possible to determine whether a recursive function terminates in a base case or whether another recursive call occurs. Non-recursive functions are simply expanded in place.

### D. Proving equivalence of terms

The equivalence proof for two terms is done in phases. A (miter) term is built to represent the equality of the two terms, and the goal of equivalence checking is to reduce the equality to true.

In some cases, simplifications at the word-level suffice to prove the equality of the two computations. If not, the next phase, bit-blasting, is applied. Bit-blasting splits word-level operations into bit-level operations; it helps regularize the structure of the two computations and prepares them for the next phase. Simplifying both before and after bit-blasting is important, because transformations obvious in one phase often are not obvious in the other. For example, it is convenient to apply commutativity of 32-bit addition at the word-level, but after the addition operations have been bit-blasted (turned into ripple-carry adders), the commutativity property is no longer easy to apply. If simplifying and bit-blasting are not enough to prove the computations equivalent, the final phase, equivalence checking, is applied to the simplified term.

A vast amount of work has been done on expression optimization and logical transformations, and many of the transformations and simplifications applied before equivalence checking are well known. The challenge is in choosing which transformations to apply (and which ones not to apply) and in what order to apply them. Seemingly minor changes in the simplifications applied or in their order can make the difference between the equivalence checking process taking a long time (or never finishing at all) and finishing quickly. Also, transformations to handle certain coding idioms (discussed later) sometimes prove to be crucial. The transformations used in our verification method have been motivated by real problems encountered in verifying block cipher code. When we found that STP was unable to discharge a proof obligation, we analyzed the failure and added transformations to handle the relevant issues. We now have a set of transformations which we believe to be robust and effective. Most of the transformations are currently performed outside of STP, but we may soon modify STP to perform them.

Block cipher implementations have several characteristics which make them difficult to verify, and we discuss these in the following sections.

### E. Handling bit manipulation

Block ciphers perform extensive bit manipulation (shifting, masking, concatenating, extracting, rotating, etc.). For example, the AESLightEngine class achieves efficiency by packing data bytes into machine words; each group of four bytes is concatenated into a 32-bit word. The bytes are repeatedly extracted from the packed representation, processed, and then repacked. A complication in verifying implementations that manipulate bits is that different programmers use different idioms to perform equivalent bit manipulations. For example, bit-vector concatenation is usually expressed by shifting one value and then combining it with the other value using an operation such as OR.

However, instead of the OR operation, the XOR operation is sometimes used. Since there is always at least one zero in every pair of bits to be combined (due to the shift), OR and XOR behave the same. Addition is also sometimes used, since the presence of zeros guarantees that there is never any carry from one bit position to the next. The use of different operations to implement concatenation means that equivalent operations may not have the same syntactic form. So our method recognizes these patterns and changes them into calls of the bit-vector concatenation operator, which provides a unique representation for this operation and best reflects the computation actually being performed. The rules to turn occurrences of OR, XOR, and addition into equivalent concatenation operations are a bit complicated; they require the presence of zeros in such a way that the combination operations never combine two ones. Yet they reflect common coding idioms and are not specific to any particular block cipher. Bit manipulation issues such as these could be handled by simply bit-blasting all word-level operations immediately. However, it is more efficient to do these transformations at the word level first, which may enable further word-level simplifications.

#### F. Handling bit rotations

Another common operation in block cipher code is bit rotation, in which bits shifted out are not dropped but rather used to fill in the positions opened up by the shift. (Since no bits are lost, the operation, like the cipher as a whole, is reversible.) Java and C programs implement rotation using other operations. A common idiom is to perform two shifts followed by a combination operation (OR, XOR, or addition, as discussed in the previous section). Again, the existence of several different but equivalent idioms requires special handling when normalizing terms.

When the rotation amount is a known constant, a rotation operation can be turned into a concatenation of two bit slices and then handled using the techniques for concatenation. However, some block ciphers (e.g., RC6) contain “variable” or “data-dependent” rotations,” in which the rotation amounts are not constants but rather depend on the inputs to the cipher [11]. Such operations cannot be easily turned into concatenations of slices because the indices in the slices and the bit-widths of the resulting values would not be constants, which most bit-vector theories require. Also, variable rotations cannot be bit-blasted, since one doesn’t know which bit will end up in which position.

So variable rotations are handled specially. The rotation idiom (two shifts and a combination using OR, XOR, or addition) is recognized and transformed into a call to a special LEFTROTATE operator. (Any right rotation can also be expressed as a left rotation.) If subsequent simplifications do not simplify the whole overarching term to true, the rotations must be expanded before equivalence checking. This can be done with if-then-else operators that test the rotation amount. However, wrapping occurrences of the rotation idiom into LEFTROTATE operations sometimes enables the entire

overarching term to be simplified down to true (as is the case for RC6), with no equivalence checking phase required.

#### G. Handling lookup tables

A common optimization in block cipher code is the use of pre-computed lookup tables to replace sequences of logical operations (e.g., XORs, shifts, masks, etc.). Such optimizations can interfere with the comparison of an optimized implementation with an unoptimized reference implementation or specification. Lookup tables are represented in our terms as array subterms with constant values for all of their elements, and our verification method can in some cases exploit patterns in the array values to rewrite table lookups into the equivalent logical operations.

Consider for example a three-bit bit-vector  $x = x_2x_1x_0$  for which one wants to compute the three bit result  $(x_2 \oplus x_1)@(x_2 \oplus x_0)@(x_1 \oplus x_0)$ . Here “@” denotes concatenation and  $\oplus$  denotes the exclusive-or operation. To compute the XOR of two of the bits of  $x$  would require several operations (a shift, an XOR, a mask, and possibly another shift to put the result into position). Instead of doing that for each bit of the result, one could simply use  $x$  as an index into the lookup table  $T$ :

$T[000]$	$=$	00000000	$T[100]$	$=$	00000110
$T[001]$	$=$	00000011	$T[101]$	$=$	00000101
$T[010]$	$=$	00000101	$T[110]$	$=$	00000011
$T[011]$	$=$	00000110	$T[111]$	$=$	00000000

where the table elements are bytes expressed in binary. This example is representative of what appears in practice in block cipher code: a lookup table in which each bit of the values encodes the XOR of some subset of bits from the table index. Our method translates such tables to logical terms when it is simple to do so. First, tables (such as  $T$ ) whose elements are more than one bit wide are blasted into tables with one-bit elements. Table  $T$  would be blasted into 8 tables, one for each column of bits; the table  $T_0$  for the least significant bits would contain the rightmost bits of  $T$ ’s values. A lookup,  $T[x]$  would be suitably transformed into the equivalent concatenation:  $T_7[x]@T_6[x]@T_5[x]@T_4[x]@T_3[x]@T_2[x]@T_1[x]@T_0[x]$ . Note that after the blasting of  $T$ , five of the resulting tables ( $T_7$  down through  $T_3$ ) contain all zeros. Clearly a lookup into a table with all zeros returns zero, and the method makes that simplification. More generally, if all the values in a table are the same, a lookup into the table is simplified to that value.

It remains to turn the nonzero single-bit tables, such as  $T_0$  above, into logical formulas. The method recognizes when table entries correspond to XORs of certain index bits, and so can transform  $T_0[x]$  into  $x_1 \oplus x_0$ . One could include other simplifications as well (e.g., identifying index bits that are ANDed in or ORed in), but the ones described above are sufficient to handle the examples that we have seen in practice (since XOR is very common in block ciphers).

#### H. Normalizing XOR terms

The XOR operation is very common in cryptography because it is easily invertible, and the verification method

normalizes nested XOR terms. It is important to handle XOR “nests” before translating to conjunctive normal form; SAT-based tools often handle XORs poorly, and the XOR nests sometime involve many operands. Since XOR is associative and commutative there are many ways to express the result of XORing several values together (i.e., there are many XOR trees of various shapes for a given set of leaves), and the verification method ensures that equivalent nests are transformed to be syntactically identical. The simplifications ensure that, after normalization, an XOR nest has the following properties: All constituent XOR operations are binary and are associated to the right. Values being XORed are sorted (by their node number in the overarching term), with constants at the front. Multiple constants are combined, and a constant of 0 is dropped (since XORing with 0 has no effect). Pairs of the same value are removed (since XORing a value with itself gives 0). Negations of values being XORed are turned into XORs with ones (since negating a bit is equivalent to XORing it with the value 1).

For example, if an XOR nest contains both the bit  $x$  and its negation, the latter will become an XOR of  $x$  with the constant 1, the 1 will get moved to the front (and perhaps combined with other constants), and the two  $x$ 's will be removed as a pair of duplicates. After normalization, two XOR nests which could be shown to be equivalent using only the properties of XOR and negation are transformed to be syntactically identical. This normalization can enable further simplifications and help the rest of the verification succeed.

In many cases, applying word-level simplifications, bit-blasting, and then simplifying again suffices to complete the verification. If not, the simplified term is passed to the final verification phase, equivalence checking. In such a case, the normalizing transformations done during the simplification phase may aid the equivalence checking phase because the two implementations have been made more similar.

## V. EQUIVALENCE CHECKING

Equivalence checking is the final phase of the verification method. Often the terms involved are large (tens of thousands of distinct nodes, even after the simplifications of the previous phases). Simply calling an off-the-shelf satisfiability solver, combinational equivalence checker, or decision procedure for bit-vectors and arrays rarely produces good results.

To prove equivalence of the large terms that represent block cipher computations one must use correspondences between internal nodes of the two computations to break down the large equivalence proof into a sequence of smaller ones. This is a well known technique in equivalence checking, and it works quite well for block ciphers and similar computations. One reason is that such computations are usually structured as a sequence of rounds. A cipher typically has a set of bits that represent its internal state, and each round modifies these bits. Different implementations may compute the rounds differently (e.g., using different optimizations or by executing operations in a different order), but the state bits almost always match up between rounds. Since computations are bit-blasted before

equivalence checking is done, differences in data packing will not prevent internal correspondences from being detected, since the nodes considered will be single bits.

In our approach, random test cases are used to identify nodes which match up between the two implementations. Clearly if any test case falsifies the top equality node, a counterexample has been found and the verification attempt has failed. Otherwise, nodes which agree in value for all test cases are considered to be “probably equal”. In practice a few dozen to a few hundred test cases usually suffice to eliminate most spurious “probably equal” pairs. (We also detect “probably constant” nodes, but such nodes rarely survive simplification.)

Pairs of “probably equal” nodes are used to split the large equivalence proof into a sequence of smaller equivalence proofs, one for each pair. When two “probably equal” nodes are proved to be actually equal, they are merged. That is, one node is chosen as the representative, and all parents of the other node are changed to depend on the representative. The proving and merging proceeds up the term, from the inputs toward the root, until the nodes for the two computations are proven equivalent and merged. When that happens the root node becomes an equality of some node with itself, which is trivially true. All this is very similar to the SAT-sweeping (also called “fraiging”) done by combinational equivalence checkers such as ABC [7].

The equivalence proofs for pairs of nodes are done by calling out to STP, a decision procedure for bit-vectors and arrays. However, the terms involved are often very large, so abstraction is used to keep the formulas sent to STP small when possible. Proofs are cut by replacing heuristically-chosen nodes with primary inputs, allowing large shared subterms to be ignored and not given to STP. Adding primary inputs for a proof results in an STP goal that is more general than the original, so if STP proves it, the original equivalence is also established. Cutting equivalence proofs in this way is a known technique, and it suffers from false negatives. When STP fails to prove the cut formula, less aggressive cuts are attempted until either STP is given a formula it can prove or it reports a counterexample for the full formula. If a counterexample is found, then the “probable equality” must in fact be false. In such a case, the nodes incorrectly thought to be equal are simply not merged, and the equivalence check continues with the next pair of “probably equal” nodes (after printing a message to the user about the failure). Such false negatives are particularly rare with block ciphers. Usually a false negative (represented as an assignment of values to the nodes along a cut) occurs because those nodes cannot actually assume that particular combination of values when the surrounding context is considered. But the nodes representing the state of a block cipher can usually take on all combinations of values.

Since verifying block cipher implementations amounts to proving combinational miter, we compared our method to ABC, a state-of-the-art hardware equivalence checker. Of course, ABC supports neither Java bytecode nor recursive functions, so we again used our term simplifier to do the symbolic simulations and unroll the specifications. To test

the effect of our word-level transformations, we refrained from performing them before calling ABC. We did have to perform transformations to get rid of all array operations (since ABC does not support them) and to bit-blast all word-level operations into simple logic gates. We didn't "charge" any of the translation time to ABC. Still, ABC took significantly longer than our tool. For "light" AES, our tool took 72 seconds total (including the symbolic simulation and specification unrolling). ABC took 385 seconds just for the equivalence check (or longer, depending on which set of simplifications we applied before calling it). For "regular" AES, our tool took 196 seconds total, but ABC took 2149 seconds for the equivalence check (or more, again, depending on which set of simplifications we performed first). For RC6 (a more challenging example), our method took 48 seconds total, and ABC took at least 1.5 days (at which point we got a "bus error").

## VI. CONCLUSIONS AND FUTURE WORK

The verification method of this paper demonstrates the feasibility of highly automated correctness proofs of block cipher implementations. The correctness results are strong (bit-for-bit equivalence with the official specification or between two implementations), and the effort and computer time required are relatively low. Thus, our method may be worth applying on a routine basis.

The techniques described in this paper should apply to implementations of block ciphers in other programming languages, machine languages, or hardware circuits, as long as one can provide appropriate translators to unroll the computations into mathematical terms. Future work might involve dealing with loops in cryptographic code in a better way, so that we can verify cryptographic hash functions on messages of arbitrary lengths.

## VII. ACKNOWLEDGMENT

This work was supported by the National Science Foundation, under the ACCURATE program (sponsor reference number CNS-0524155-003). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the view of the National Science Foundation.

## REFERENCES

- [1] Federal Information Processing Standard Publication 197. <http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf>.
- [2] C. L. Berman and L. H. Trevillyan. Functional comparison of logic designs for vlsi circuits. In *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, Nov. 1989.
- [3] Galois Connections. Cryptol. <http://www.cryptol.net/>.
- [4] J. Duan, J. Hurd, G. Li, S. Owens, K. Slind, and J. Zhang. Functional correctness proofs of encryption algorithms. In G. Sutcliffe and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2005)*, volume 3835 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, December 2005.
- [5] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV '07)*, Berlin, Germany, July 2007. Springer-Verlag.
- [6] Andreas Kuehlmann and Florian Krohm. Equivalence checking using cuts and heaps. In *Design Automation Conference*, pages 263–268, 1997.

- [7] Alan Mishchenko, Satrajit Chatterjee, Robert Brayton, and Niklas Een. Improvements to combinational equivalence checking. In *ICCAD '06: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design*, pages 836–843, New York, NY, USA, 2006. ACM.
- [8] J Moore. Proving Theorems about Java and the JVM with ACL2. <http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-02/index.html>.
- [9] N.I.S.T. and Lawrence E. Bassham III. The Advanced Encryption Standard Algorithm Validation Suite (AESAVS). <http://csrc.nist.gov/groups/STM/cavp/documents/aes/AESAVS.pdf>.
- [10] Lee Pike, Mark Shields, and John Matthews. A verifying core for a cryptographic language compiler. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*, pages 1–10, New York, NY, USA, 2006. ACM.
- [11] Ronald L. Rivest, M.J.B. Robshaw, R. Sidney, and Y.L. Yin. The RC6 Block Cipher. <http://theory.lcs.mit.edu/~rivest/rc6.pdf>.
- [12] Diana Toma and Dominique Borrione. SHA formalization. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '03)*, July 2003.
- [13] Sean Weaver. Equivalence checking. <http://gauss.ececs.uc.edu/Courses/C702/Weaver/ec.01.08.07.ppt>.
- [14] Xiang Yin, John C. Knight, Elisabeth A. Nguyen, and Westley Weimer. Formal verification by reverse synthesis, to appear in SAFECOMP 2008.
- [15] Qi Zhu, Nathan Kitchen, Andreas Kuehlmann, and Alberto Sangiovanni-Vincentelli. Sat sweeping with local observability don't-cares. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 229–234, New York, NY, USA, 2006. ACM.