Proving Functional Equivalence of two AES Implementations using Bounded Model Checking [*]

Hendrik Post and Carsten Sinz
Institute for Theoretical Computer Science
University of Karlsruhe, Germany
{post,sinz}@ira.uka.de

Abstract

Bounded model checking—as well as symbolic equivalence checking—are highly successful techniques in the hardware domain. Recently, bit-vector bounded model checkers like CBMC have been developed that are able to check properties of (mostly low-level) software written in C. However, using these tools to check equivalence of software implementations has rarely been pursued. In this case study we tackle the problem of proving the functional equivalence of two implementations of the AES crypto-algorithm using automatic bounded model checking techniques. Cryptographic algorithms heavily rely on bit-level operations, which makes them particularly suitable for bit-precise tools like CBMC. Other software verification tools based on abstraction refinement or static analysis seem to be less appropriate for such software. We could semi-automatically prove equivalence of the first three rounds of the AES encryption routines. Moreover, by conducting a manually assisted inductive proof, we could show equivalence of the full AES encryption process.

1 Introduction

Increasing the reliability of software can be done in different ways, e.g. by elaborate testing methods. If an even higher level of confidence is required, verification approaches are employed. Over the last years, considerable progress has been made in transferring fully automatic verification techniques from academic research to industrial practice. Promising techniques for verification include abstract interpretation, model checking in combination with abstraction refinement, and bounded model checking.

Abstraction based software verification approaches assume that significant performance gains can be achieved by "simplifying" the program under test such that it represents only aspects required for the verification task at hand. This idea has been reported to work well for many control-command oriented software systems, e.g. device drivers [1, 21]. The applicability of this approach, however, is dependent on the ability to find aspects that can be safely abstracted.

Whereas for high-level software abstraction seems to be a natural choice, low-level software often sacrifices abstractable concepts in favor of efficient implementations. Examples for these tradeoffs can be found among embedded device software that is often still implemented in C. Software implementations of ciphers, like the Advanced Encryption Standard (AES) [9], fall into this category, too. The AES standard is a symmetric block cipher using the formerly named Rijndael algorithm. In order to provide efficient algorithms for block encryption, AES relies on bit-sensitive hardware operations for encryption and decryption. To provide a high level of security, every bit of the ciphertext must be dependent on every bit of the original text and every bit of the key. Both characteristics imply that data-slicing as well as data-abstraction do not perform well.

In such cases, where abstraction and high-level state space pruning may fail, bit-sensitive techniques provide a good alternative. One example for such a technique is Bounded Model Checking which has been adapted to work directly on software implementations. The usual problem is that bit-sensitive verification approaches are expected not to scale well, especially in the presence of unbounded loops. State-space explosion is a commonly cited reason that prohibits the sound and complete verification of real world software systems.

This work addresses the question whether and how a software bounded model checking tool like CBMC [6] can be used to verify equivalence of two implementations of the AES standard. Even though the above software implementations provide a substantial challenge to verification methods, we were able to obtain equivalence proofs for

[*]This work was supported in part by the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

two of three parts of the algorithm. Solved as well as unsolved instances are available for download from our website[1]. In the following we describe how such results can be achieved (Section 3). In Section 4 results are presented and discussed. Section 5 expands the discussion towards general challenges that software verification approaches face in practice. Lessons learned are also given in this section. It should be noted that the *decision procedure*, i.e. software bounded model checking as implemented by CBMC, is completely automatic. Nevertheless, the decomposition of the verification task and specification of data mappings and proof obligations must be generated manually.

2 Tools and AES Implementations

Software Bounded Model Checking is one technique for verifying safety properties of finite state software systems. The approach as well as the implementation is reviewed in the next section. Additionally the application domain, AES, will be introduced to explain the common structure among both implementations.

2.1 Software Bounded Model Checking with CBMC

CBMC [6] is a bounded software model checking tool for ANSI-C programs. Its implementation works similar to bounded model checking tools for hardware verification. All memory locations that may be addressed in a C program of bounded length are modelled by finite bit-vectors. The bound gives a maximum number of loop iterations and recursion unwindings that may occur on each unwound path. Recursion and loops are inlined such that the overall bound is not violated. The resulting program has a finite number of statements and therefore the number of possibly addressed memory locations is also finite. All operations are translated into bit-vector transitions. The well-known single static assignment form (see e.g. [8])enables the creation of stateless bit-vector formulas which are further simplified to pure boolean formulas in CNF. A boolean satisfiability decision procedure then decides whether safety properties hold for all possible finite length executions. For all experiments in this paper CBMC is only used to generate CNF formulas. The formulas are then solved using Minisat2 [12].

CBMC has built-in checks for several common runtime errors. However, custom properties can be specified in the C program using `assert` statements. `assume` statements enable custom restrictions on the program state.

In order to check equivalence of two C functions that are side-effect free both functions are called in a wrapper

```
// first implementation
function int A(int input)
{ ... }
// second implementation
function int B(int input)
{ ... }

void miter() {
// miter function that checks for equivalence
   int input_A  = nondet_int();
   int input_B  = nondet_int();
// inputs must be equal
   assume(input_A == input_B);
// execute functions sequentially
   int result_A  = A (input_A);
   int result_B  = B (input_B);
// verification obligation
   assert(result_A == result_B);
}
```

Figure 1. Checking functional equivalence with CBMC is straight-forward iff functions are side-effect free and the decision procedure handles the state-explosion.

program. Prior to the execution of the functions, input parameters are defined to be arbitrary (non-deterministic), but equal. After execution of both functions, explicit or implicit outputs are checked for equivalence using the `assert` directive.

Figure 1 shows a typical example for a wrapper program implementing the equivalence condition. Note that functions of different implementations must use different namespaces when checked for equivalence.

2.2 AES

The AES standard, formally named Rijndael algorithm, is a symmetric block cipher with variable block and key length. Block cipher refers to the fact that a text is encrypted using a small block with constant size at a time. In standard encryption mode each processing of a block is independent of the encryption or decryption of other blocks. The size of the blocks is predefined to be 128,192, or 256 bits. The key is defined to hold 128, 192, or 256 bits. In the following experiments key and text block sizes are limited to 128 bits.

The AES implementations differ greatly from control-command driven program code as device driver implementations:

1. AES follows a strict execution order, i.e. for given bit-sizes the exact execution trace can be determined at compile-time. CBMC detects sound unwinding bounds automatically. The amount of used stack memory is consequently also bounded. Heap allocation is not used.

[1]`http://www-sr.informatik.uni-tuebingen.de/~post/aes/index.html`

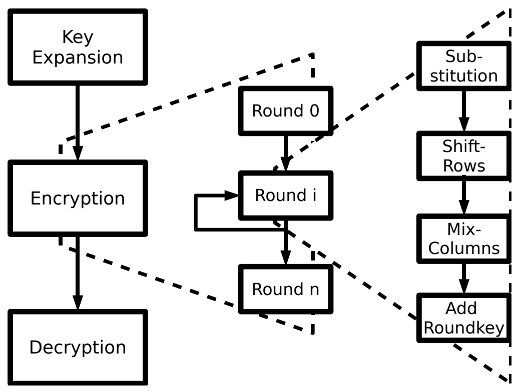

Figure 2. AES is a round-based encryption cipher. Encryption and decryption are iterated $n = 10..14$ times. Each round contains 4 phases which are executed in reverse order for decryption.

2. AES relies on hardware supported bit-vector operations like XOR and SHIFT.
3. Large quantities of constant tables are used.
4. Cipher-texts are substituted in a non-linear way .
5. All bits occurring in the cipher and plain text are involved in the encryption and decryption process. Thus they cannot be sliced away in the verification equations (*Diffusion* phases MixColumn and ShiftRows).

AES is a round based algorithm using 10,12, or 14 iterative rounds. The ciphertext output from the last round is taken as an input for the following round. The key is expanded and distributed such that for each round a distinct round key is generated. A schema of the AES flow is given in Figure 2.

Implementations of AES have to provide implementations for key expansion, encryption and decryption.

2.2.1 Reference Implementation (RI)

Barreto and Rijmen created a reference implementation that is available for download[2]. The following functions provide implementations for the three basic phases:

- `int rijndaelKeySched (word8 k[][], int keyBits, int blockBits, word8 rk[][][])`
- `int rijndaelEncrypt (word8 a[][], int keyBits, int blockBits, word8 rk[][][])`
- `int rijndaelDecrypt (word8 a[][], int keyBits, int blockBits, word8 rk[][][])`

[2]http://www.iaik.tugraz.at/Research/krypto/AES/old/˜rijmen/rijndael /rijndaelref.zip

Table 1. AES phases are implemented by the following functions and data-structures.

AES	Reference Implementation	Mike Scott's Impl.
S-Box	`S`	`fbsub`
Inverse S-Box	`Si`	`rbsub`
Key expansion	`rijndaelKeySched`	`gkey`
Encryption	`rijndaelEncrypt`	`encrypt`
Decryption	`rijndaelDecrypt`	`decrypt`

`k` denotes the original key whereas `rk` denotes the expanded version of the key, i.e. an array of keys for the respective rounds (round keys). The bit-parameter denote the number of bits used for each block respectively for the key, i.e. 128 for our experiments. The formal parameter `a` refers to the plaintext or ciphertext.

2.2.2 Mike Scott's Implementation (MSI)

Though several C implementations exist, we have chosen the one by Mike Scott[3]. The implementation is written in ANSI-C without any fractions of inline assembly code. The structure of the implementation is roughly similar to the reference implementation.

MSI provides the following function interfaces:

- `void gentables(void)`
- `void gkey(int nb,int nk,char *key)`
- `void encrypt(char *buff)`
- `void decrypt(char *buff)`

`gentables` performs several precomputations that must be executed prior to the encryption, decryption and key-generation. `gkey` implements the key-expansion. Table 1 relates AES phases with function names of both implementations.

3 Preparations

Checking the equivalence of two implementations requires mapping inputs from one implementation to the other, as, e.g., data formats may be different. In general, we also need a way to check that the outputs of the two implementations are the same for equivalent inputs. However, in our case, the outputs of both algorithm are bit-wise equal, so no mapping is required here.

Another challenge is the following: although CBMC can handle almost all aspects of the C language, it has some problems with special constructs like complex type conversions. These also have to be dealt with in a manual preprocessing step in order to use CBMC. More specifically, we had to add code transformations to some functions that pass parameters by reference:

[3]Available under: ftp://ftp.compapp.dcu.ie/pub/crypto/rijndael.c

- Formal reference parameters are refactored such that global variables are used directly, e.g., the function `encrypt(char* buff)` becomes `encrypt(void)` and all references to `*buff` are preplaced by referencing the global buffer variable. For the given implementations this transformation can easily be applied as the formal reference parameters always refer to the same object (singleton).

- Reference parameters to subarrays of multidimensional arrays are preplaced by a reference to the parent array. Moreover, an additional index parameter is added to the function's parameter list that allows to compute the correct address using index operations.

The above changes affect slightly more than 50 lines of code in the RI. However, each individual change is trivial. The correctness of these changes is confirmed by runtime-testing and by typechecking of a compiler.

3.1 Synchronizing Inputs

RI and MSI use many common constant tables and matrices. One example is the S-box that provides one source of non-linearity in the algorithm. In cases where the AES standard defines values of constants we merge tables and arrays from both implementations. Additionally, the computation of the look-up tables in MSI can be removed. Note that this computation of, e.g., discrete logarithm constants, is deterministic, so the correctness of the removal of `gentables` can be easily verified by one execution of both algorithms. The definitions that are used in the following experiments stem from the RI. Instead of renaming references to tables in the MSI code, preprocessor macros are used to redirect read accesses to RI tables and arrays. One example is the macro `#define fbsub S`. `fbsub` is the MSI array that encodes the S-box constants. The macro implies that any access of the form `fbsub[i]` is transformed to `S[i]`. The information which arrays correspond to each other has to be extracted from the program structure and the AES standard.

Encryption, decryption and key-expansion operate on two parameters: the key and the text buffer. In the RI two-dimensional arrays are used to store the 128 bit blocks. MSI uses one-dimensional arrays of 8 bit cells for the text and of 32 bit cells for the round keys. The possible number of different mappings between the encodings of the round keys is large, but code inspection and parallel execution of MSI and RI allowed to find the correct mapping after a few tries.

The mapping of different input encodings is given in Figure 3. For the purpose of ensuring that both implementation get similar inputs the mapping had to be encoded in C. As presented in the initial example, `assume` statements are used to express equality of memory cells. The full encoding of the mapping is given in Figure 4. The embedding of the synchronization for encryption and decryption analysis is shown in Figure 5 b).

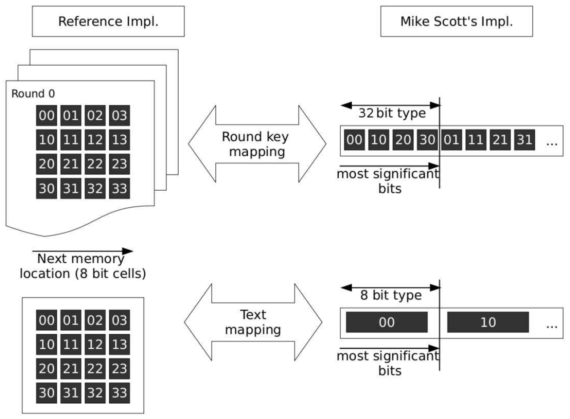

Figure 3. RI and MSI use different memory layouts for the encoding of key and texts. In order to synchronize inputs the mappings had to be reengineered and expressed via `assume` statements (cf. Subsection 3.1).

a)

```
for (i = 0; i != 4; i++) {
  for (j = 0; j != 4; j++) {
    key[i][j] = nondet_char();
// initialize the RI text buffer
    text[i][j] = nondet_char();
// initialize the MSI text buffer
    text_ms[j*4+i] = text[i][j];
  }
}
```

b)

```
BYTE tmp[4]; WORD res;
for (r = 0; r != 11; r++) {
  for (j = 0; j != 4; j++) {
     for (i = 0; i != 4; i++) {
        tmp[i] = rk[r][i][j];
     }
     res = pack(tmp);
// initialize MSI forward key fields
(*)  fkey[r*4+j]=res;
  }
}
```

Figure 4. a) Inputs, i.e. key and text buffers, have to be synchronized. b) Decryption and encryption additionally require the synchronization of rounds keys. The function `pack` encodes 4 bytes into one 32 bit word. `rk` and `fkey` denote the round keys used by RI, respectively MSI.

4 Verification Results

This section contains details of all performed experiments. The general setup of the different experiment stages are illustrated in Figure 5. Runtimes and information about generated queries for the SAT-solver are presented in Table 2.

In the following equivalence checks for all three parts of AES are discussed separately.

4.1 Key Generation

The layout of the equivalence check for key generation is depicted in Figure a). Note that equivalence checking on the key generation requires a mapping between different bits of the encoded round key array, i.e. the output of the key expansion. The mapping is depicted in Figure 3. After providing the mapping the verification was straightforward: The only variable input is the key itself which is non-deterministically initialized and passed to both functions.

The proof obligation is that the generated round keys are equal. With CBMC such a condition can be expressed similar to Figure 4 b), where the `(*)` line is transformed to express a proof obligation instead of a synchronizing assignment:

```
(*)      assert(fkey[r*4+j]==res);
```

Both decryption and encryption require round keys as well as a text of 128 bits as input. We have shown the round key generation to be equivalent. Therefore, round keys generated by RI can be used as input for both implementations .

4.2 Encryption

The system layout for the equivalence checks for the encryption routines is shown in Figure b) and c). The complexity of the verification task is increased incrementally: At first (EN1r) we execute a normal key generation followed by one round of encryption for both algorithms. Afterwards we checked the outputs to be equal. Then, the number of rounds is iteratively increased to up to 4 until a timeout is produced (EN2r,EN3r and EN4r).

For the 128 bit sizes, AES defines that ten rounds have to be performed. Inlining encryption steps ten times resulted in a SAT instance that could not be solved within twelve hours. In order to obtain sound verification results for encryption, an inductive schema was used: The base case consists in the proof that both algorithms always get equal inputs, i.e. round keys (KEY). The inductive step encodes the fact that if both algorithms are equal up to the i-th round, then, they produce equal results after round $i+1$. The proof of the inductive step (ENloop) has a smaller runtime than a two round encryption (EN2r). One round encryption (EN1r) has a significantly smaller runtime than the inductive step because AES defines the first round to require less phases than any other round.

ENzz2r encodes a two-round encryption with deterministic (zero filled) text and key bits. Hence, the proof of equivalence does not involve any (relevant) arbitrary variables. So, ENzz2r roughly estimates the amount of time used for constant propagation and two-round equivalence checking. It is notable that the runtime of ENzz2r is more than 20 times smaller than the runtime of EN2r. This result indicates that the runtime is indeed dominated by solving the encryption equations and not by plain constant propagation of the zero filled input blocks. It is also notable that using concrete key and text bits a result for ten-round encryption could be achieved (ENzz10r).

4.3 Decryption

All decryption experiments follow the same layout as the corresponding encryption ones (cf. Figure b) and c)).

The structure of the AES cipher is symmetric. One would thus expect that repeating the above experiments for the decryption phase would yield similar results. In contrast to this expectation decryption equivalence checking did involve harder problems (cf. Table 2).

It is significant that all decryption instances show much larger sizes than their corresponding encryption counterparts. Why these instances provide harder benchmark is subject to speculation. However, we suggest the following explanations:

- MSI uses a different order in which decryption is performed. This results in a loss of structural similarities in the verification equations , which decreases the amount of reusable component encodings.

- MSI generates optimized backward round keys `rkey` from forward round keys. Synchronizing inputs for MSI therefore also requires the construction of the MSI-forward keys. Reverse keys are then generated using MSI-code. The extra lines of code add several additional loops and many new bitfields to the program. As software bounded model checking uses unwinding and single static assignment forms, the differences could be exponentially enlarged.

DE1r is a proof of concept that at least one round decryption can be proved equivalent. DE2r and DE3r SAT instances could be generated. Both confirm a blowup in their problem sizes as well as CBMC generation times. A two-round decryption run with concrete key and text bits could also be completed.

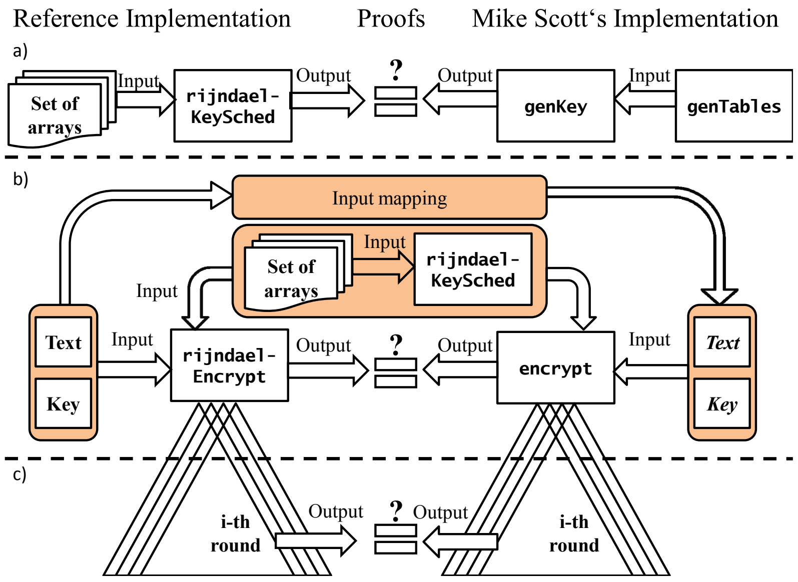

Figure 5. Components of the equivalence proofs.

As the RI and the MSI show significant differences in their decryption loop implementation a formal induction proof could not be performed. The overall high runtimes hamper attempts to correctly establish inductive proofs. DEloop shows runtimes and problem sizes for an incorrect induction step, i.e. the claim does not hold. Runtimes for a correct invariant are probably much higher.

Table 3 provides statistics about the amount of source code and source code changes that were involved in the experiments. In the following the runtimes and problem sizes are discussed.

4.4 Runtimes and Problem Sizes

The runtimes of CBMC and Minisat2 are listed in the first columns of Table 2. Encryption related experiments are denoted by the prefix EN. Decryption experiments have the prefix DE.

EN vs. DE runtimes. Decryption checking runtimes are much higher than encryption checking runtimes. One possible explanation on program level is that for decryption forward and reverse keys must be computed. The later are computed from the forward keys which are only needed for the encryption. Especially for a small number of rounds the overhead of computing twice as many keys might explain a difference in runtimes. The increase in runtimes matches the increase in variables and clauses needed to encode the problem in CNF.

EN1 vs. EN2, EN3, EN4 runtimes. The relative increase between EN1 and EN2 is much higher than for other experiments. This discrepancy is caused by the fact that AES requires the first encryption and the last decryption round to consist of only the addition of the round keys. Normal rounds include 4 computationally intensive steps instead of one. This effect is also visible in the differences between EN and DE runtimes because DE1r,DE2r and DE3r do not include the reduced last round whereas EN1r,EN2r and EN3r profit from this irregularity.

ENzz vs. EN runtimes. As expected, zero-initialized text and key buffers lead to a significant runtime improvement compared to equivalence checks for arbitrary, but equal keys for both implementations. The high runtime of ENzz10r is caused by the high number of variables and clauses: Minisat2 needs to propagate the initial input to output variables.

CBMC emitts the number of program assignments and sliced program assignments as additional information. For the sake of completeness they are also shown in Table 2.

The last columns contain the number of variables and clauses taken from the DIMACS CNF encoding that is the input to Minisat2. It is not surprising but notable that the largest solveable instance, ENzz10r, is solved quite fast even though it involves more than 2 million variables. This is due to the fact that it does not contain any relevant free variables as text and key bits are determined.

Table 2. Statistics of different experiments (using Minisat2 as SAT-Solver). (S) denotes a satisfiable instance, all others are unsatisfiable. (TO) refers to a timeout (more than 12h). Instances that have a zz postfix were generated using a zero filled key and text array. Some instances were manually sliced (**).

	CBMC[s]	SAT[s]	Assignm.	Sliced	Variables	Clauses
KEY	25.44	76.40	13,958	5,641	79,145	475,049
EN1r	21.20	2.60	15,577	9,489	34,665	221,485
EN2r	37.22	1,313.45	16,471	7,472	267,541	1,573,693
EN3r	51.68	3,274.87	17,365	7,756	500,385	2,925,901
EN4r	69.59	(TO)	18,259	8,040	733,229	4,278,109
ENloop	344.63	21.47	10,502	4,654	276,374	1,598,822
ENzz2r	37.27	5.99	16,488	7,521	253,125	1,486,261
ENzz10r	166.95	954.36	23,740	9,831	2,116,005	12,303,925
DE1r	20.15	1.92	15,645	9,515	34,658	220,870
DE2r(**)	69.07	(TO)	10,659	4,035	886,030	4,821,654
DE3r(**)	172.49	(TO)	20,131	8,152	1,737,370	9,422,438
DEloop(S)	1,446.81	2,868.36	26,632	8,953	4,244,263	22,604,459
DEzz2r	85.86	15.14	17,905	7,730	883,094	4,805,802
DEzz10r	1,679.54	(TO)	35,849	11,498	7,693,814	41,612,074

5 Experiences

In the last section raw results were presented. This section concentrates on the insights gained while performing the case study. A brief proposal for the integration of the technique into industrial development processes is given in Subsection 5.3.

Two high-level lessons could be learned: At first, we faced problems finding correct mappings between different structures of the two implementations, Moreover the verification of the decryption failed due to the fact that an invariant or mapping could not be found. A resulting consequence is that we propose that verification must be performed in cooperation with the software developers. This experience is also brought forward in [11]. Secondly, state space explosion is a still a major obstacle for functional equivalence checking. Nevertheless even data-sensitive verification tasks with non-linear operations can be established using state-of-the-art SAT-solving tools.

5.1 Preparation - Verification Cycles

The first stage of any software verification project is the program preparation and the specification task description. Program prepration could mean that the program needs to be syntactically transformed, manually sliced, abstracted or prepared for manual induction. Each of these tasks is error-prone for several reasons: The most obvious one is that the internal workings of the software are not evident for the verification practitioner who is not the developer. Another reason is that manual abstraction or transformations must ensure that the truth of the verification property is not changed.

Table 3. Indication of program and modification statistics. (*) The RI includes two files that contain numerical constants.

Characteristics	Sizes
MSI LOC	386
RI LOC (*)	381 + 85 + 64
Decrypt Miter LOC	63
Encrypt Miter LOC	64
Keygeneration Miter LOC	32
Macros to map MSI to RI tables	5

This means that the verification practitioner needs to understand *prior* to the verification task why certain properties hold—respectively why certain aspects are irrelevant to the property.

So in practice software verification is not a one-pass process of specification or preparation followed by a single invocation of the verification engine. A better model is an iterative process that involves reengineering of specifications, transformations and the program until no evident errors occur. Each iteration involves an invocation of the backend. Thus, the overall performance of the cycle is much more time-consuming than the runtime given for a single verification run.

Iterative processes require a termination condition: *When should the iterative refinement of program and specification stop?* Above we defined that the termination condition is fulfilled when no obvious errors in transformations and specifications occur. A better solution requires proof explanations which will be discussed in the following subsection.

5.2 Proof Explanation

CBMC generates SAT-instances which are solved by a SAT-solver. If the instance is satisfiable a model is generated and CBMC constructs a concrete trace in the program from the model. Hence, for satisfiable properties,i.e. instances where the property does not hold, the verication practitioner may check if the trace forms a real program error. Another outcome is that the specification or some program transformations could be wrong—i.e. a false positive bug report.

What if CBMC reports that a property holds? If a SAT-solver supports proof generation (which Minisat2 does not support at the moment) this proof could theoretically provide an explanation for the correctness of the formula. Of course, the bit-level proof also represents a proof in the program semantics. However, it seems completely impossible to understand a SAT-proof for a formula with a few million variables. It remains an open problem how a better explanation can be generated. Otherwise, all correctness proofs remain subject to possible specification, program transformation or verification engine implementation errors.

A more formal representation of a verification task is given by

$$P \wedge T \models S$$

P represents the set of statements that the program defines. S denotes the specifications whereas T denotes performed program transformations (e.g. the mapping of input parameters in equivalence checking). No matter whether the above statement holds, the result is always dependent on the correct formulation of S and T. A proof explanation seems the only way to ensure that both S and T are indeed as intended. Notably the current state of the art technique for increasing confidence in specifications and transformations is fault injection and—to a limited extent—testing.

5.3 Industrial Integration

The results obtained in the experiments show that source code bounded model checking is a powerful tool for systematically analyzing and comparing computationally intensive functions. Nevertheless, the provided data resembles an upper bound of the current performance of this technique. Given these bounds and the insight that equivalence checking greatly depends on structural similarity, we derive one domain where source code bounded model checking might be used successfully: regression proofs for low-level functions that have been changed with respect to coding styles or optimizations.

MISRA-C [20] is such a coding standard that is widely required for embedded software development in the automotive domain. MISRA-C provides a set of rules that restrict the use of C language constructs. At one of the later stages in the development process, a compliance analysis is performed. If functions or lines of code violate MISRA-C rules the code must be changed. As the functions were already tested on unit and system level it is desireable to automatically prove that the changes did not introduce new flaws in the program. Otherwise some of the unit and system tests would have to be rerun.

In cooperation with the automotive supplier Robert Bosch GmbH, Germany, we tested whether software model checking is capable to prove functional equivalence on the level of C functions. Initial experiments indicate that integration of this technique works well in industrial practice. In cases where the function was unintentionally changed with respect to its functional behavior, CBMC detected the flaw within seconds without having to specify test cases.

5.4 Related work

Matsumoto et al. [18] present an equivalence checking method for C descriptions. One of the cases they consider is an AES implementation. Their cases as well as the employed methods differ greatly from this case study. Instead of checking two entirely different implementations, the authors insert a small number of textual changes in the C description. For the AES implementation the only change employed is the replacement of 4-XOR operations by 2-XOR operations. Equivalence checking is then performed by symbolic execution focussed on equivalence class abstraction. The application of the proposed method is limited as input descriptions must have an isomorphic control-flow graph.

Using SAT solvers to tackle cryptographic problems is an active area of research. Massacci and Marraro analyzed the Data Encryption Standard by manually encoding the algorithm in propositional logic [17]. They were able to break the encryption and find the encryption key for up to three encryption rounds, but not for the complete DES. Compared to our work, their approach requires a manual (and potentially error-prone) encoding of the crypto algorithm, whereas our method works directly on the C implementation. Further work on using SAT solvers for cryptography includes the analysis of hash functions [19, 10]

Clarke and Kroening [5] briefly state that equivalence for a C and VHDL implementation of the DES crypto algorithm can be proved using CBMC. However, the authors provide very few details on the performed experiment. A notable difference is that DES is by far less complex than AES. Additionally it is unclear how the VHDL and C implementation relate to each other—if the VHDL code is a direct implementation of DES that has been derived from the C program this could have eased the proof significantly.

Kim et al. [14] present a recent case study where CBMC is used to unit test functional requirements of a flash memory device driver.

Many software model checking case studies deal with abstract interface specifications. Ball et al. [1] perform a case study on windows device drivers searching for violations of finite state interface protocols. The properties checked are limited to function call triggered state transitions. Post and Küchlin [21] present a similar case study using bounded model checking for Linux device drivers. MOPS is another light-weight model checking tool that has checked several higher-level specifications in Linux software [4].

The above three case studies deal to a large amount with control-command oriented program code which is typical for the functions performed by device drivers. AES functional consistency checking shows inverse statistics. The code base is smaller but the amount of lines of code and percentage of bits that are relevant to the verification property is much higher.

A recent trend in software verification is the application of multiple abstraction techniques: Abstract interpretation [7] static analysis tools are known to provide insights on programs using various abstractions on variable and operation domains. A recent case study on Avionics software is presented by Delmas and Souyris [11]. Other techniques apply hard-coded abstractions (e.g. summaries) to reduce the complexity that arises when checking complex software systems (cf. Engler and Ashcraft [13]). One seminal result is that (predicate) abstraction can also be computed and refined in a lazy manner. The latter approach is named counter example guided abstraction refinement and is for example implemented in software verification tools like MAGIC [3]. Bryant et. al recently proposed abstraction based decision procedures for bit-vector arithmetic [2].

Our technique may also profit from techniques like SAT Sweeping [15] or proof reuse [16] to achieve better performance.

6 Summary

The overall contribution of this work is the demonstration that functional equivalence checking of real-world software is feasible using standard academic software bounded model checking tools. Most of the verification work is done automatically and does not require user-interaction. However, manual preparation is still needed to a certain degree. Most notable in our case is the semi-automatic induction we had to perform to decompose the problem into manageable parts. Using novel invariant generation methods—as presented in [**?**] or in [**?**]—a fully automatic verification of this inductive part might become possible in the near future.

In addition, our work provides benchmark instances to support the development and testing of future verification engines on the basis of real world applications.

We would like to thank Alexander Kaiser and Thomas Gorges for supporting the industrial integration.

References

[1] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys Conf., Proc.*, pages 73–85. ACM Press, 2006.

[2] R. E. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. A. Brady. Deciding bit-vector arithmetic with abstraction. In O. Grumberg and M. Huth, editors, *TACAS*, volume 4424 of *Lecture Notes in Computer Science*, pages 358–372. Springer, 2007.

[3] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *25th Intl. Conf. on Software Engineering (ICSE), Proc.*, pages 385–395. IEEE Computer Society, 2003.

[4] H. Chen and D. Wagner. MOPS: an infrastructure for examining security properties of software. In *9th ACM Conf. on Computer and Communications Security (CCS), Proc.*, pages 235–244. ACM Press, 2002.

[5] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311. IEEE Computer Society Press, January 2003.

[6] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[7] P. Cousot. Abstract interpretation. *ACM Comput. Surv.*, 28(2):324–328, 1996.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 25–35, New York, NY, USA, 1989. ACM.

[9] J. Daemen and V. Rijmen. The block cipher rijndael. In J.-J. Quisquater and B. Schneier, editors, *CARDIS*, volume 1820 of *Lecture Notes in Computer Science*, pages 277–284. Springer, 1998.

[10] D. De, A. Kumarasubramanian, and R. Venkatesan. Inversion attacks on secure hash functions using satsolvers. In *Proc. of the 10th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT 2007)*, pages 377–382, 2007.

[11] D. Delmas and J. Souyris. Astrée: From research to industry. In H. R. Nielson and G. Filé, editors, *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 437–451. Springer, 2007.

[12] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[13] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *19th ACM Symp. on Operating Systems Principles (SOSP), Proc.*, pages 237–252. ACM Press, 2003.

[14] M. Kim, Y. Kim, and H. Kim. Ieee/acm int. conf. on automated software engineering (ase). In *Int. Conf. on Automated Software Engineering (ASE), Proc., (to appear)*. IEEE Computer Society Press, September 2008.

[15] A. Kuehlmann. Dynamic transition relation simplification for bounded property checking. In *2004 International Conference on Computer-Aided Design (ICCAD'04)*, pages 50–57, 2004.

[16] J. Marques-Silva. Interpolant learning and reuse in SAT-based model checking. *Electr. Notes Theor. Comput. Sci.*, 174(3):31–43, 2007.

[17] F. Massacci and L. Marraro. Logical cryptanalysis as a SAT problem. *Journal for Automated Reasoning*, 24(1/2):165–203, 2000.

[18] T. Matsumoto, H. Saito, and M. Fujita. An equivalence checking method for C descriptions based on symbolic simulation with textual differences. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, E88-A(12):3315–3323, 2005.

[19] I. Mironov and L. Zhang. Applications of SAT solvers to cryptanalysis of hash functions. In *Proc. of the 9th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT 2006)*, pages 102–115, 2006.

[20] MISRA C Working Group. *MISRA-C:2004 - Guidelines for the use of the C language in critical systems*. The Motor Industry Software Reliability Association, 2004.

[21] H. Post and W. Küchlin. Integration of static analysis for linux device driver verification. In J. Davies and J. Gibbons, editors, *Integrated Formal Methods (IFM), 6th Intl. Conf., Proc.*, volume 4591 of *LNCS*, pages 518–537. Springer-Verlag, 2007.