Exploiting Refactoring in Formal Verification

Xiang Yin, John Knight, Westley Weimer
Department of Computer Science, University of Virginia
{xyin,knight,weimer}@cs.virginia.edu

Abstract

In previous work, we introduced Echo, a new approach to the formal verification of the functional correctness of software. Part of what makes Echo practical is a technique called verification refactoring. The program to be verified is mechanically refactored specifically to facilitate verification. After refactoring, the program is documented with low-level annotations, and a specification is extracted mechanically. Proofs that the semantics of the refactored program are equivalent to those of the original program, that the code conforms to the annotations, and that the extracted specification implies the program's original specification constitute the verification argument. In this paper, we discuss verification refactoring and illustrate it with a case study of the verification of an optimized implementation of the Advanced Encryption Standard (AES) against its official specification. We compare the practicality of verification using refactoring with traditional correctness proofs and refinement, and we assess its efficacy using seeded defects.

1. Introduction

Developing software that is sufficiently dependable for critical applications is a difficult challenge. A desirable technology for helping to meet that challenge is formal verification. Unfortunately, although formal verification has proven effective, it is not widely used. In part, this is because of pragmatic difficulties.

In previous work, we introduced the Echo approach to formal verification [14, 17]. The goal that we have for Echo is to make formal verification of software more practical. We seek an approach that works seamlessly with existing software development techniques, that can be applied routinely and with reasonable effort, and that requires only average skill.

By formal verification we mean the establishment of a proof based on logical inference (as opposed to model checking) that a given program is a correct implementation of a given specification. Different aspects of a specification are sometimes dealt with separately in formal verification. For example, verification of functionality is often separated from verification of timing in real-time software. Our focus in this paper is on verification of functionality.

A factor that frequently limits formal verification of functionality is the complexity of the subject software. Efforts to build software that is compact, efficient, and highly functional tend to produce software systems that, in principle, could be verified, but for which the human effort involved and the detail management required make formal verification either unattractive or infeasible.

To deal with this problem, Echo includes a mechanism that we refer to as *verification refactoring*. The concept is to refactor software that was developed by conventional means using semantics-preserving transformations to produce a functionally equivalent version for which formal verification is practical. The transformations that are applied are selected solely to facilitate the major verification proofs and each is proven to be semantics preserving.

In this paper, we discuss the concept and mechanism of verification refactoring. As part of the refactoring process, we introduce the use of software complexity metrics as a tool for guiding the refactoring process. We present preliminary assessment data from a case study of the verification of an optimized implementation of the Advanced Encryption Standard (AES) against its official specification. We also present the results of an experiment in which we seeded defects into the implementation to determine the difficulty developers might face when locating defects that cause formal verification to fail.

2. Existing Approaches to Verification

Existing approaches to verification fall basically into three categories: correctness proofs, refinement, and model checking and static analysis.

Correctness proof, for example the weakest precondition approach, tries to establish the theorem that

978-1-4244-4421-2/09/$25.00

when a program's precondition is satisfied, its postcondition will be satisfied after execution. The main difficulty that arises with it is complexity. Although machine assistance has been developed, the details can easily overwhelm whatever machine resources are available, even for relatively small programs. The issue is not just the cumulative detail for the program, but also the complexity of individual predicates associated with elaborate or intricate source statements.

Refinement based approaches such as the B Method [1] have been created in response to practical difficulties with correctness proofs. Software development by refinement involves the transformation of an abstract specification to a concrete implementation by a series of refinement transformations. The output of each transformation is proved to imply the input.

Creating a proof along with the program to which it applies is a laudable goal. However, the goal restricts the exploration of alternatives during software development and leads to the following limitations:

- Many existing software development techniques cannot be used because software development is constrained by the simultaneous proof development.
- If changes to an existing program are required to meet performance goals, the whole refinement path needs to be revisited so as to update the proof.

These limitations essentially make refinement approaches either impractical or undesirable for the vast majority of software developments.

To achieve necessary levels of assurance for crucial applications, testing is usually not feasible, and so mechanical analysis, where possible, is an attractive alternative. In addition to correctness proof and refinement, several other verification techniques have been developed, such as model checking and static analysis, to try to facilitate mechanical verification. Although such techniques scale quite well and have been applied successfully, their analysis targets only certain properties. For crucial applications, functional verification is highly desirable.

3. The Echo Verification Approach

We present a brief summary of the Echo approach. Further details are available elsewhere [14, 17].

At the heart of Echo verification is a process that we refer to as *reverse synthesis* in which a high-level, abstract specification (that we refer to as the *extracted* specification) is synthesized from a low-level, detailed specification of a system. Verification then involves two proofs: (1) the **implementation proof**, a proof that the source code implements the low-level specification correctly; and (2) the **implication proof**, a proof that the extracted specification implies the original system specification from which the software was built. Each of these proofs is either generated automatically or mechanically checked, and each can be tackled with separate specialized techniques and notations.

The Echo approach imposes no restrictions on how software is built except that development has to start with a formal system specification, and developers have to create the low-level specification documenting the source code. There are no limitations on design or implementation techniques nor on notations that can be used. The present instantiation of Echo uses: (1) PVS [12] to document the system specification and the extracted specification; (2) the SPARK subset of Ada [3] for the source program; and (3) the SPARK Ada annotation language to document the low-level specification. In the current instantiation, the proof that the extracted specification implies the system specification is created using the PVS theorem prover, and the proof that the low-level specification is implemented by the source code is created by the SPARK Ada tools. The extracted specification is created by custom tools.

4. Motivation for Verification Refactoring

Informally, by verification refactoring we mean the transformation of a program in such a way that the functional semantics of the program (but not necessarily the temporal semantics) are preserved and verification is facilitated. The reverse synthesis process in Echo makes extensive use of verification refactoring, and it is a critical part of the way in which Echo is made more broadly applicable. In this section, we discuss the motivation for verification refactoring in terms of the difficulties that it helps to circumvent in the two Echo proofs.

Significant effort in software development goes into making sure that the software is adequately efficient. The result of this effort is careful treatment of special cases, compact data structures and efficient algorithms, with the inevitable introduction of complexity into the control- and data-flow graphs. Much of the difficulty in formal verification results from the complexity of the source program. One of the reasons for the use of verification refactoring is to reduce this complexity.

A second reason for the use of verification refactoring is to align the structure of the extracted specification with the structure of the system specification. This alignment permits the implication proof to be structured as a series of lemmas and allows an efficient

overall proof structure.

The transformations used and the mechanism of their selection is different for the two proofs, and so we discuss each separately in this section.

4.1. Support For The Implication Proof

The implication proof is the proof that the extracted specification implies the original specification from which the program was written. In principle, if the software is indeed a correct implementation of the specification, then it is always possible to construct such a proof. The challenge in Echo, however, is to make the construction of the proof relatively routine.

The feasibility of this proof rests in large measure on the form, content and structure of the extracted specification. Echo uses several techniques to synthesize this specification [17], but the key in Echo to making the proof practical lies in a technique that we refer to as *architectural and direct mapping*. This technique rests on the hypothesis that the high-level architectural information in a specification is frequently retained in the implementation. We have no experimental evidence to support this hypothesis, but our rationale for believing it is discussed in an earlier paper [17].

Architectural and direct mapping provides the basis of the implication proof. The structure of the proof is based on the specification architecture. The basic approach that we use is to try to match the static function structure of the extracted specification to the original specification, and to organize the proof as a series of lemmas about the specification architecture.

With this approach to proof, the closer the extracted specification's architecture comes to that of the original specification, the higher the chance of the proof being completed successfully and in a reasonable time. The transformations that are selected to apply to the source program are those which will align the extracted specification's architecture more closely with that of the original specification.

4.2. Support For The Implementation Proof

The implementation proof is the proof that the implementation implies the low-level specification. In the prototype Echo system, the implementation proof is carried out using the SPARK Ada toolset. The preferred approach to developing SPARK Ada software is to use *correctness by construction* [6]. In correctness by construction, the SPARK Ada tools are often able to complete proofs with either no or minimal human intervention. The proof process is repeated as the software is constructed thereby ensuring that each refinement leaves the software amenable to proof.

By contrast in Echo, since there are no restrictions on development techniques, the SPARK Ada tools frequently fail when they are applied to software after development is complete. The low-level design of software that is not developed using correctness by construction is unlikely to be in a form suitable for proof. The reasons are many but, as with the implication proof, they typically fall under the heading of complexity introduced to achieve some specific design or performance goals.

The difficulties with the SPARK proof system take one of three forms: (1) the required annotations for function pre- and post-conditions can be many dozens of lines long, lengths that are impractically complex for humans to write; (2) the implementation proof exhausts available resources, usually memory, even though the SPARK tools are quite efficient and typically adequate for proofs that are needed for correctness by construction; and (3) the verification conditions sometimes are sufficiently complex that they cannot be discharged automatically, and human guidance becomes necessary.

Verification refactoring addresses all three of these difficulties without limiting the development process. Because verification refactoring does not need to maintain any aspect of efficiency, any transformation that addresses the three types of difficulty can be used.

5. The Refactoring Process

5.1. Definition of Refactoring

The Echo verification argument relies upon refactoring, and so it is essential that there be a precise definition of refactoring and a mechanism for ensuring that refactoring complies with this definition in practice. Since Echo is verifying functional behavior, we make the following three simplifying assumptions: (1) the source program terminates; (2) refactoring does not preserve the execution time of the program; and (3) refactoring need not preserve the exact sequence of intermediate program states as long as the initial state and final state are unchanged. Assumption (3) also implies that floating-point arithmetic accuracy is not guaranteed to be preserved and that the semantics of non-thread-safe programs are not preserved. The transformation from program `P` to program `P'` is *semantics preserving* if, given the same initial state, both `P` and `P'` will terminate and generate the same final state.

We need to be able to prove that any given transformation is semantics preserving, and, in order to do so for the general case, we define the semantics of the

elements we need to model the transformation in PVS. For example, systems states are modeled as mappings between identifiers and values, statement blocks and subprograms are modeled as transitions between states, and pre- and post-conditions are predicates over states. For each generalized transformation, we use the PVS theorem prover to discharge the following theorem:

```
init_state(P) = init_state(P')
    => final_state(P) = final_state(P')
```

We have developed a preliminary library of transformations for which the necessary properties have been proved. Similar libraries of semantics preserving transformations exist in the domains of compilation, software maintenance, and reverse engineering. We have included some common transformations in our library, but few existing transformations can be adapted because they have different goals. Compilation transformations, for example, are usually targeted at performance improvement. Ours are designed to reduce the complexity and size of verification conditions, and so frequently reduce software's efficiency.

Here we itemize some of the refactorings that we have developed and discuss how each affects the goal of verification. Due to space limitation, we do not include examples for each of them.

Rerolling loops. A sequence of repeated statement blocks that can be differentiated by a certain parameter can be converted into a loop based on that parameter. For example, if the parameter is an integer taking sequential values, we can turn the statements into a simple for-loop:

S_1; S_2; …; S_n; ⟺

```
for i in range 1..n loop S(i) end loop;
```

Rerolling unrolled loops allows generated verification conditions to be simplified by recovering the loop structure and permitting the introduction of loop invariants, especially when the repeated statement block is large.

Moving statements into or out of conditionals. Moving statement blocks into or out of conditional statements provided no side effects will result can help to simplify execution paths and to reveal certain properties. An example would be the following if statement block. S_1 has no effect on conditional `B`:

S_1; if B then S_2 else S_3 end if; ⟺

if B then S_1; S_2 else S_1; S_3 end if;

Splitting procedures. Long procedures usually result in verbose and complex verification conditions. By splitting a procedure into a set of smaller sub-procedures, the verification conditions become vastly simpler and easier to manage.

Adjusting loop forms. Loops are frequently defined to promote efficiency and ease of use. Adjustment of the loop parameters can facilitate verification by, for example, allowing loop invariants to be inserted more easily thereby simplifying verification conditions.

Reversing inlined functions or cloned code. Reversing inlined functions involves identifying cloned code fragments and replacing them with function definitions and calls. Function definitions can be provided by the user or be derived from the code. This transformation aligns the code structures with the specification and removes replicated or similar verification conditions so as to facilitate proof. Furthermore, by reversing the inlining of functions, if an error is identified in a particular inlined function, only that function needs to be reverified rather than all of the inlined instances.

Separating loops. Loops that combine operations can be split so as to simplify the associated loop invariants.

Modifying redundant or intermediate computations or storage. These transformations modify the program by adding or removing redundant or intermediate storage or computation. This can facilitate proof by: (a) storing extra but useful information; (b) shortening the verification condition by removing redundant or intermediate variables; or (c) merely tidying the code so as to facilitate understanding and annotation of the code.

All the above refactorings and associated proofs are for general programs. We discuss the use of these refactorings and the results of applying them in our case study in section 6.

5.2. Applying Refactoring

Our process for applying verification refactoring in practice is shown in Figure 1. A semantics-preserving transformation from the library is selected by the user (or suggested automatically), and the transformer then checks the applicability of the selected transformation mechanically and applies it mechanically if it is applicable. When all of the selected transformations have been applied, a metrics analyzer collects and analyzes the code properties of the transformed code, and presents the complexity metrics to the user. If the metric results are not acceptable, or if they are acceptable but later verification proofs cannot be established, the process goes back to refactoring and more transformation are performed.

The role of the source-code metrics is to give the user insight into the likely success of the two Echo proofs. We hypothesize that the metrics we use are an indication of relative complexity and therefore of likely verification difficulty, and we present some support for this hypothesis in the case study.

Verification refactoring cannot be fully automatic in the general case, because recognizing effective transformations requires human insight except in special cases. Furthermore, some software, especially domain-specific applications, might require transformations that do not exist in the library. In such circumstance, the user can specify and prove a new semantics-preserving transformation using the proof template we provide and add it to the library.

To facilitate exploration with transformations, if the user has confidence in a new transformation, the semantics-preserving proof can be postponed until the transformation has been shown to be useful or even until the remainder of the verification is complete.

In most cases, the order in which transformations are applied does not matter. Clearly, however, when two transformations are interdependent, they have to be applied in order. A general heuristic is that those transformations that change the program structure and those that can vastly reduce the code size should be applied earlier.

We are not aware of any circumstances of their application in which a transformation would have to be removed, and we make no explicit provision for removal in the current tools and process. In the event that it becomes necessary, removing a transformation is made possible by recording the software's state prior to the application of each transformation.

All the user activities, especially the design and selection of transformations, have to be mechanically checked, and these two activities need to be supported by automation to the extent possible. The transformer is implemented using the Stratego/XT toolset [4]. Stratego checks the applicability of the selected transformation, and carries it out mechanically using term rewriting. We use the PVS theorem prover as the transformation proof checker and provide a proof template. When the user specifies a new transformation, an equivalence theorem will be generated automatically, and the user can discharge it interactively in the theorem prover.

To our knowledge, there is no verification complexity metric available that could guide the user in selection of transformations, and so we present a hybrid of metrics to the user for review using a commercial metric tool [2], the SPARK Examiner, and our own analyzer. The metrics include:

Element metrics. Lines of code, number of declarations, statements, and subprograms, average size of subprograms, logical SLOC, unit nesting level, and construct nesting level.

Complexity metrics. McCabe cyclomatic complexity, essential complexity, statement complexity, short-circuit complexity, and loop nesting level.

Verification condition metrics. The number and size of verification conditions, maximum length of verification conditions, and the time that the SPARK tools take to analyze the verification conditions.

Specification structure metrics. A summary and comparison of the architectures of the original and the extracted specifications to suggest an initial impression of the likely difficulty of the implication proof.

Interpretation of the metrics is subjective, and we do not have specific values that would give confidence in the ability of the PVS theorem prover to complete the implication proof.

We developed the following heuristics to both select transformations and determine the order of application: (1) transformations that depend on each other are applied in order; (2) transformations that impact the major sources of difficulty, such as code and VC size, are applied first; (3) transformations that affect global structure are applied earlier and those that affect local structure are applied later; and (4) refactoring proceeds until all proofs are possible.

In practice, if specification extraction or either of the proofs fails to complete, or if either proof is unrea-

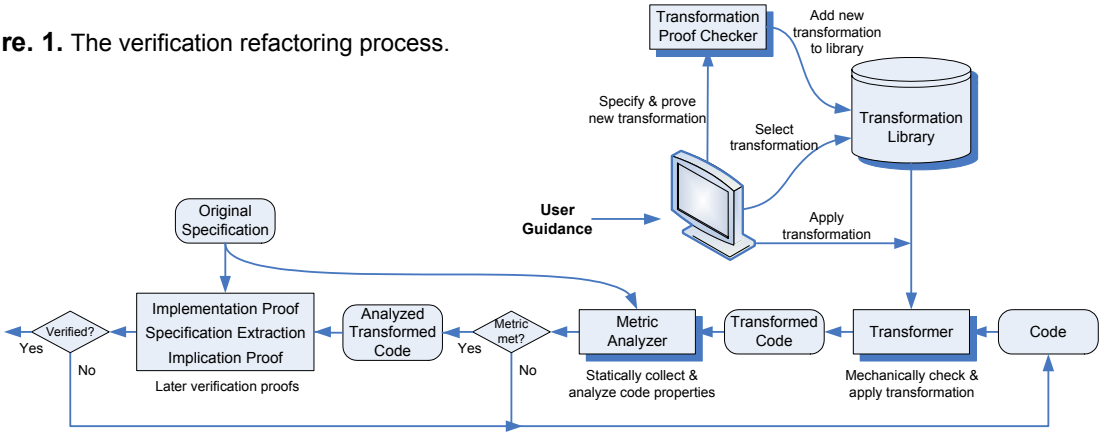

Figure. 1. The verification refactoring process.

sonably difficult, the user returns to refactoring and applies additional transformations.

6. Evaluation of Refactoring

In order to obtain an initial assessment of the efficacy and utility of verification refactoring, we undertook the verification of a non-trivial program that we did not develop. The issues that affect the efficacy and utility of verification refactoring include: (1) the ease with which developers can select transformations; (2) the ease with which developers can add domain specific transformations and prove them to be semantics preserving; (3) whether selected transforms do facilitate the necessary proofs; and (4) whether refactoring impedes development in some way.

Issues 1, 2, and 3 are tied closely to our use of metrics, since we anticipate the values of metrics being the basis for developers' decisions. We sought to determine: (1) the impact on metrics of individual types of refactoring and of series of refactorings; and (2) the values of the metrics for software that was amenable to proof and refactorings that were suggested by the values of metrics. Our experience with refactoring in the verification of the subject application is the focus of this section and provides information about the first three issues. In section 7, we address the fourth issue.

6.1. The Advanced Encryption Standard

The subject of study was the Advanced Encryption Standard, and we used artifacts from the National Institute of Standards and Technology (NIST). In previous work [17], we conducted a study of the general feasibility of an earlier version of Echo in which we verified only part of AES. In the work described here, we verified the functional correctness of the complete AES implementation. The AES artifacts that we used were:

FIPS 197 specification. The Federal Information Processing Standards Publication 197 [8] specifies the AES algorithm, a symmetric, iterated block cipher. The specification is mostly in natural language with mathematical statements and pseudo code for some algorithmic elements.

ANSI C implementation. Developed by Rijmen et al. [7], this optimized implementation is written in ANSI C. It is 1258 lines of code and contains several optimizations to enhance its performance.

These two artifacts were written independently of this project by others, and so there were no constraints on the development process imposed by the subsequent application of Echo.

6.2. AES Verification

We developed a formal version of the FIPS 197 specification in PVS and translated the ANSI C implementation into SPARK Ada, the notations used in the current Echo instantiation. The PVS specification is 811 lines long, excluding boilerplate constant definitions. The SPARK Ada implementation (1365 lines without annotations) was created by translation of the C statements into corresponding Ada statements.

The verification of AES employed the complete Echo process: (1) a series of refactoring transformations were applied; (2) the final refactored version was documented using the SPARK Ada annotation language; (3) the code was shown to be compliant with the annotations; (4) a high-level specification was extracted from the refactored, annotated code; and (5) the extracted specification was shown to imply the original specification.

6.2.1. Verification Refactoring. The AES implementation employs various optimizations (including implementing functions using table lookups, fully or partially unrolling loops, and packing four 8-bit bytes into a 32-bit word) that improved performance but also created difficulties for verification. For instance, the SPARK tools ran out of resources on the original program because the unrolled loops created verification conditions that were too large.

We applied 50 refactoring transformations in eight categories. Of those 50, the following 38 transformations from six categories were selected from the prototype Echo refactoring library (the number after the category name is the number of transformations applied in that category): rerolling loops (5); reversing inlined functions or cloned code (11); splitting procedures (2); moving statements into or out of conditionals (3); adjusting loop forms (4); modifying redundant or intermediate computations (2); and modifying redundant or intermediate storage (11). The rationale and use of these transformations are discussed in the next section. In addition to these transformations, we also added two new transformation categories for AES:

Adjusting data structures (2). 32-bit words were replaced by arrays of four bytes, and sets of four words were packed into states as defined by the specification. Constants and operators on those types were also redefined accordingly to reflect the transformations.

Reversing table lookups (10). Ten table lookups were replaced with explicit computations based on the documentation and the precomputed tables removed.

Both of these two added transformation types were driven by the goal of reversing documented optimiza-

tions and matching the extracted specification to the original specification. The final refactored AES program contained 25 functions and was 506 lines long.

6.2.2. Complexity Metrics Analysis. Using the heuristics mentioned earlier, we selected and ordered transformations to use with AES. Rather than examining the effects of each transformation separately, we grouped the transformations into the following 14 blocks: (1) loop rerolling for major loops in the encryption and decryption functions; (2) reversal of word packing to use four-byte arrays; (3) reversal of table lookups; (4) packing four words into a state; (5) reversal of the inlining of the major encryption and decryption functions; (6) reversal of the inlining of the key expansion functions; (7) moving statements into conditionals to reveal three distinct execution paths followed by procedure splitting; (8) adjustment of loop forms; (9) reversal of additional inlined functions; (10) loop rerolling for sequential state updates; (11) procedure splitting; (12) adjustment of intermediate variables; (13) adjustment of loop forms; and (14) additional procedure splitting.

Blocks 7-11 were for the subprogram that set up the key schedule for encryption, and blocks 12-14 were for the subprogram that modified the key schedule for decryption. As well as the main transformations, each block of transformations involved smaller transformations that modified redundant or intermediate computations and storage.

As part of determining whether further refactoring was required, we periodically attempted the proofs and determined the source-code metrics. Some of the results of the effect of applying the transformations on the values of the metrics are shown in Figure 2. The histograms show the values of different metrics after the application of the 14 blocks of transformations where block 0 is the original code.

As the transformations were applied, the primary element metric, code size, dropped from over 1365 to 412. The average McCabe cyclomatic complexity also declined, dropping from 2.4 to 1.48. Statement complexity, essential complexity, etc. also declined. There is no evidence that these complexity metrics are related to verification difficulty, but their reduction suggests that the refactored program might generate less verbose VCs and be easier to analyse.

Since we would not undertake full annotation until refactoring was complete, we had no way to assess the feasibility of the proofs. To gain some insight, we set the postconditions for all subprograms to *true* for each version of the refactored code, generated verification conditions (VCs) using the SPARK examiner, and simplified the generated VCs using the SPARK simplifier. We measured the number of VCs, the size of VCs, the maximum length of VCs, and the time that the SPARK tools took to analyze the code. These data did not necessarily represent the actual proof effort needed for the implementation proof, but they were an indication.

The times required for analysis with the SPARK tools after the various refactorings are shown as Figure 2(c). Some blocks are shown with no value because the VCs were too complicated to be handled by the SPARK tools. After the first loop rerolling at block 1, the tools completed the analysis but took 7 hours and 23 minutes on a 2.0 GHz machine. At block 2 with word packing reversed, the analysis again became infeasible. Analysis by the SPARK tools became feasible again by block 8 after we had adjusted the loop forms. The required analysis reached 1 minute 42 seconds for the final refactored program.

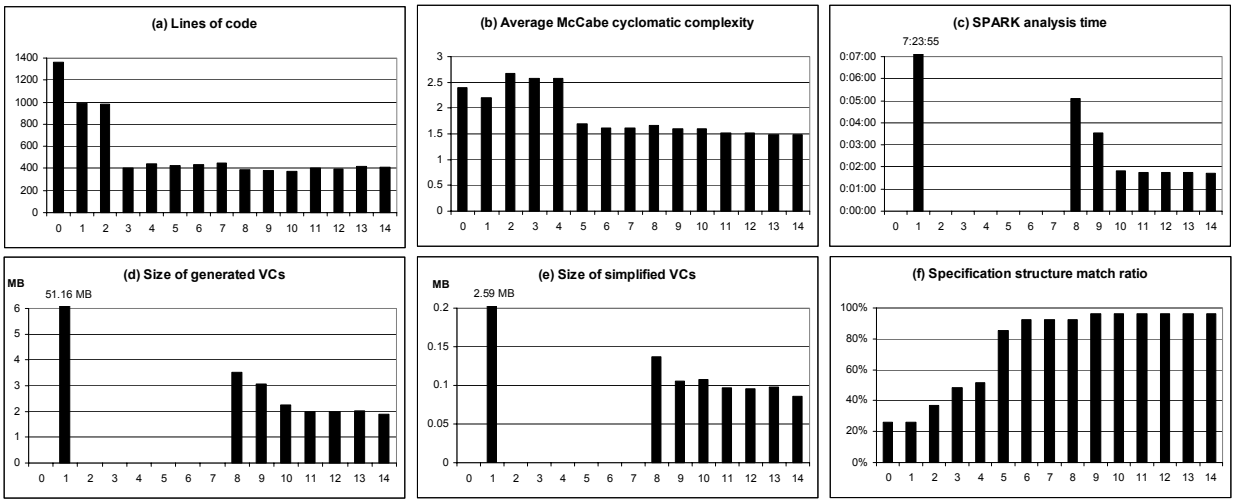

Figure. 2. Metric analysis with AES verification refactorings

In block 1, 51.16 MB VCs were generated and 2.59 MB were left after simplification. For the final refactored code, 1.90 MB VCs were generated and 86 KB were left after simplification (Figs 2(d) and 2(e)).

The simplified VCs were those that needed human intervention to prove. After block 1, the maximum VC length was over 10,000 lines. In the final refactored code, the maximum was 68 lines. When the implementation annotation was complete, the maximum length of VCs needing human intervention was 126 lines.

We extracted a skeleton specification from the code after applying each block of transformations. These specifications were skeletons because they were obtained before the code had been annotated. We compared the structure of the skeleton extracted specification with that of the original specification by visually inspection and evaluated a *match-ratio* metric. This is defined as the percentage of key structural elements—data types, operators, functions and tables—in the original specification that had direct counterparts in the extracted specification. We hypothesize that this measure is an indication of the likelihood of successfully establishing the implication proof.

The values of the match ratio are shown in Figure 2(f). The ratio increased gradually from 25.9% to 96.3% as the transformation blocks were applied. There is only a small increase in its value after the block 8 transformations were applied, and the implication proof could have been attempted at that point. However, since the time required for the SPARK analysis was still declining, we chose to continue refactoring until all metrics stabilized.

6.2.3. Implementation Proof. After refactoring, the code was examined and annotated manually. The actual numbers of annotations are shown in Table 1:

Table 1: Annotations in implementation proof

Type	Lines
Preconditions	8
Postconditions	123
Loop Invariants & Assertions	54
Proof Functions, Proof Rules, & Other	32

The implementation proof was carried out using the SPARK Ada toolset. A total of 306 VCs were generated, of which 86.6% were discharged automatically in 145 seconds on a 2.0 GHz machine. 15 out of 25 functions had all VCs discharged automatically. The remaining VCs required quite straightforward manual intervention, usually involving either the application of preconditions or induction on loop invariants. The interactive proof process for each remaining VC was finished within a few minutes by a single individual who has a good level of SPARK Ada experience.

Throughout the proof process, the length of the VCs remained completely manageable. No difficulties were encountered in reading or understanding them, or in manipulation of them with the SPARK tools.

6.2.4. Implication Proof. The extracted specification (in PVS) produced by the Echo specification extraction tool was 1685 lines long. It was much larger than the original specification because the implementation contained tables for multiplication in the $GF(2^8)$ field which were not present in the original specification.

When typechecking the extracted specification, the PVS theorem prover generated 147 Type Correctness Conditions (TCCs), of which 79 were discharged automatically by the theorem prover in 23.5 seconds on a dual 1.0 GHz machine and the remaining 68 were all subsumed by the proved ones.

As a result of verification refactoring, the architecture of the extracted specification was sufficiently similar to the architecture of the original specification that we were able to identify the matching elements easily. To prove the extracted specification implied the original one, we created an implication theorem using a general process that is part of Echo [17].

There were 32 major lemmas in the implication theorem. Type checking of the implication theorem resulted in 54 TCCs, 29 of which were discharged automatically in 4.2 seconds on a dual 1.0 GHz machine and 25 were subsumed by the proved ones.

In most cases, the PVS theorem prover could not prove the lemmas completely automatically. However, the human guidance required was short and straightforward, typically including expansion of function definitions, introduction of predicates over types, or application of extensionality. In some cases, introducing other previously proved supporting lemmas and structuring the proof as cases were required. Each lemmas was established and proved interactively in a few minutes (thus the implication theorem discharged).

7. Refactoring and Defect Detection

When using formal verification, defects in the subject programs are revealed by a failure to complete the proof. Proof failures always present the dilemma that either the program or the proof could be wrong. This dilemma is present with any method, including testing.

Verification refactoring might make the dilemma worse or introduce other forms of difficulty in identifying defects. In order to investigate this issue, we seeded defects into the original AES implementation and then

determined the effect of each defect on verification. We present the results of that experiment in this section.

7.1. The Seeding Process

The seeding process was done by randomly choosing a line number and performing a change in the code. Each defect in the program was a change in either: (a) a numeric value; (b) an array index; (c) an operator (for computation or predicate); (d) a variable or table reference; or (e) a statement or function call.

These types of defect are not equivalent to those introduced by programmers. However, they do reflect common errors that might be introduced, and there is some evidence that simple seeded defects share important properties with actual defects [9].

Code and therefore the defects are closely tied into the annotations that document the low-level specification. The defective code could be annotated so as to either describe its desired behavior rather than its actual behavior, or vice versa. We used both scenarios in this experiment and evaluated them separately.

7.2. Defect Location

There are three stages in the proof process that could expose defects in the code:

Verification refactoring. A defect could change the code such that it did not match a particular transformation template and the transformation could not be applied. For example, a defect in only one iteration of an unrolled loop rather than in all interactions would make loop rerolling inapplicable.

Implementation proof. Any inconsistency between the code and the annotations would be detected by the SPARK Ada tools. An inconsistency could arise because of a defect in either or both.

Implication proof. Defects in the code but with consistent annotations, or postcondition annotations that are not strong enough, would cause the implication theorems to be unprovable and so would be detected by the implication proof.

7.3. Experimental Results

We seeded 15 defects, three defects of each basic type, one at a time into the AES implementation, and then we ran the Echo verification process twice for each defect. In the first (setup 1), we assumed that the defects were caused by misunderstandings of the specification when implementing the code, and the annotations corresponded to the functional behavior of the code. In the second (setup 2), we assumed that the defects were introduced by implementation errors, and the annotations corresponded to the high-level specification. The results are shown in tables 2 and 3.

Table 2: Defect detection for setup 1

Verification Stage	Defects Caught	Defects Left
Initial state		15
Verification refactoring	4	11
Implementation proof in SPARK	2	9
Implication proof in PVS	8	1

For setup 1, most defects were caught during the implication proof since the annotation matched the code. The two defects that were caught in the implementation proof were found during the proof of exception freedom because they caused possible out-of-bound array references. The remaining defect that was not caught at any stage was benign. We discuss it later.

Table 3: Defect detection for setup 2

Verification Stage	Defects Caught	Defects Left
Initial state		15
Verification refactoring	4	11
Implementation proof in SPARK	10	1
Implication proof in PVS	0	1

For setup 2, most defects were caught during the implementation proof since the annotation did not match defective code. The remaining defect was the same benign defect.

In both setups, verification caught the same 14 seeded defects. The remaining (benign) defect changed an array of keys. The length of the array had been set to accommodate the maximum number of rounds in the case of a 256-bit key length. However for key lengths of 128 bits or 192 bits, the last several entries in the array were not used in the computation. This was purely an implementation decision, and the specification did not impose any restrictions. Thus, for shorter key lengths these entries could be allowed to have arbitrary values without affecting functional correctness.

Echo does require that the developer annotate the code, and, whenever there is an unprovable proof obligation, the user has to determine whether it is the result of a defect in the code or the annotations. However, the use of architectural and direct mapping in the creation of the extracted specification means that the location of defects can be restricted to the function that cannot be proved. In the AES case, each function is quite small and manageable after verification refactoring, making defect location quite simple.

8. Related Work

Retrieval of abstract specifications from source code through formal transformations has been reported in the reverse-engineering domain [5, 16]. The goal is to improve the structure of poorly-engineered code and to facilitate further analyses.

Paul et al. [13] are developing an approach to the determination of how refactorings affect the verifiability of a program. Their focus is object-oriented design, and the goal is to see whether a syntactic change can make more properties amenable to analysis.

Smith et al. [15] have developed an infrastructure for verifying properties of block ciphers, including AES, and they have verified AES implementations in Java byte code. They noted different representations between the specification and the implementation, and provided transformation functions between the two.

Kuehlmann1 et al. [10] have developed an approach called transformation-based verification for sequential verification of circuit-based designs. The approach uses structural transformation that relocates registers in a circuit-based design representation without changing its actual input-output behavior, to increase the capacity of symbolic state traversal. Verification refactoring adopts similar idea to transform the target being verified, but for software.

9. Conclusion

Refactoring deals with many of the issues that limit the applicability of formal verification including unworkably large verification conditions and the rigid development process necessary for refinement.

We have demonstrated the efficacy and utility of refactoring by verifying a moderate-sized program written by others and not designed for verification. The refactoring process was guided by a set of complexity metrics that helped both select transformations and determine when the program was likely to be amenable to proof. Off-the-shelf verification was impossible using conventional tools, but the addition of refactoring made the task both feasible and straightforward. In an experiment using seeded defects, we have also demonstrated that locating defects in software for which verification is being attempted is fairly straightforward, even when verification refactoring is being applied.

10. Acknowledgements

Work funded in part by NASA grants NAG-1-02103 & NAG-1-2290, and NSF grant CCR-0205447.

11. References

[1] Abrial, J.R., *The B-Book: Assigning Programs to Meanings*, Cambridge University Press, 1996.

[2] Adacore, *GNAT Metric Tool*, http://www.adacore.com

[3] Barnes, J., *High Integrity Software: The SPARK Approach to Safety and Security*, Addison-Wesley, 2003.

[4] Bravenboer, M., K.T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/XT 0.16. A Language and Toolset for Program Transformation", *Science of Comp. Progr.*, 2007.

[5] Chung, B. and G.C. Gannod, "Abstraction of Formal Specifications from Program Code", IEEE 3rd Int. Conference on Tools for Artificial Intelligence, 1991, pp. 125-128.

[6] Croxford, M., and R. Chapman, "Correctness by construction: A manifesto for high-integrity software", CrossTalk, *The Journal of Defense Soft. Engr*, 2005, pp. 5-8.

[7] Daemen, J. and V. Rijmen, "AES Proposal: Rijndael. AES Algorithm Submission", 1999.

[8] FIPS PUB 197, "Advanced Encryption Standard", National Institute of Standards and Technology, 2001.

[9] Knight J.C., and P.E. Ammann, "An Experimental Evaluation of Simple Methods For Seeding Program Errors", ICSE-8: Eighth Int. Conf. on Soft. Engr, London, UK, 1985.

[10] Kuehlmann, A., and J. Baumgartner, "Transformation-based verification using generalized retiming", Computer Aided Verification, Paris, France, 2001, pp. 104-117.

[11] Liskov, B. and J. Wing, "A Behavioral Notion of Subtyping", *ACM Transactions on Programming Languages and Systems*, 16(6):1811--1841, 1994.

[12] Owre, S., N. Shankar, and J. Rushby, "PVS: A Prototype Verification System", CADE 11, Saratoga Springs, NY, 1992.

[13] Paul, J., N. Kuzmina, R. Gamboa, and J. Caldwell, "Toward a Formal Evaluation of Refactorings", Proc. of The Sixth NASA Langley Formal Methods Workshop, 2008.

[14] Strunk, E.A., X. Yin, and J.C. Knight, "Echo: A Practical Approach to Formal Verification", FMICS05, Lisbon, Portugal, 2005.

[15] Smith, E., and D. Dill, "Formal Verification of Block Ciphers, A Case Study: The Advanced Encryption Standard (AES)", Stanford University.

[16] Ward, M., "Reverse Engineering through Formal Transformation", *The Computer Journal*, 37(9):795-813, 1994.

[17] Yin, X., J.C. Knight, E.A. Nguyen, and W. Weimer, "Formal Verification By Reverse Synthesis", the 27th SAFECOMP, Newcastle, UK, September 2008.