AXE: AN AUTOMATED FORMAL EQUIVALENCE CHECKING TOOL FOR PROGRAMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Eric Whitman Smith

June 2011

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

David Dill, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Dawson Engler

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John Mitchell

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost Graduate Education

Abstract

This dissertation describes Axe, an automated formal verification tool for proving equivalence of programs. Axe has been used to verify real-world Java implementations of cryptographic operations, including block ciphers, stream ciphers, and cryptographic hash functions. Axe proves the bit-for-bit equivalence of the outputs of two programs, one of which may be a formal, mathematical specification. To do so, Axe relies on a novel combination of techniques from combinational equivalence checking and inductive theorem proving. First, the loops in some programs can be completely unrolled, creating large loop-free terms. Axe proves the equivalence of such terms using a phased approach, including aggressive word-level simplifications, bit-blasting, test-case-based identification of internal equivalences, and "sweeping and merging." For loops that cannot be unrolled, Axe uses execution traces to detect loop properties, including loop invariants for single loops and connection relationships between corresponding loops. Axe proves such properties inductively. In many cases, synchronizing transformations must be performed to align the loop structures of the programs being compared; Axe can perform and verify a variety of these transformations.

Acknowledgments

I would first like to thank my advisor, David Dill, for all of his help and advice during my time at Stanford, including many in-depth discussions while I was implementing Axe. He has been a great advisor, always helping me to see the big picture and to ask the hard questions. His feedback was invaluable to the successful completion of Axe and to my growth as a researcher.

Thanks also to my committee members, John Mitchell, Dawson Engler, and Alex Aiken, for helpful feedback and mind-opening discussions, and to Robert Tibshirani for serving as my committee chair.

I would like to thank Matt Kaufmann, J Moore, Bob Boyer, and the ACL2 community for the excellent ACL2 system, on which Axe is built. I am also grateful to Vijay Ganesh for developing STP, which is an integral component of Axe.

I have had the good fortune to work with many excellent colleagues during my career, and I would like to thank the folks at Rockwell Collins, AMD, the University of Texas, and Kestrel Institute for their advice and support during my Ph.D.

Finally I'd like to thank my friends and family for putting up with me and helping keep me sane even during the busiest times. Most importantly, I'd like to thank my parents, for giving me life and love. Everything good I do is ultimately because of them.

[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 author and do not necessarily reflect the view of the National Science Foundation.]

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Cryptography is the science of securely dealing with information. Cryptographic techniques are used, for example, to encrypt messages for storage and transmission, to authenticate their origin, and to protect messages against corruption. Cryptography plays an important role in many critical systems, including in applications vital to national security, network privacy, e-commerce, and electronic voting. Thus, cryptographic code is worth getting right.

Cryptographic systems are often built using small algorithmic "building blocks," such as block ciphers, stream ciphers, and cryptographic hash functions. The correctness of large systems usually depends critically on the correctness of these components; an implementation error or deliberately inserted flaw in a cryptographic building block may compromise the security of the entire system that contains it.

Validation of cryptographic building blocks is almost always done by testing. For example, the National Institute of Standards and Technology (NIST) [51] has validated hundreds of implementations of the AES block cipher [5]. Each implementation is repeatedly executed on different inputs, and the results are checked. Such validation is required of any cryptographic system that deals with sensitive U.S. government data.

Unfortunately, exhaustive testing of even small cryptographic components is typically not feasible, because they have vastly too many possible inputs. Indeed, any cryptographic algorithm with so few inputs that complete, exhaustive testing were possible would likely be insecure (breakable by a brute force attack). In some sense, then, secure cryptographic building blocks are "inherently untestable," and implementation flaws may go undetected.

Because cryptographic systems are so important, and because they may be the target of determined attackers, implementations of cryptography are usually evaluated in an "adversarial setting" in which it is assumed that a skilled attacker with significant resources is trying to break the system. In such a setting, a failure rate of even 1 in 2^{64} – which would be considered very low in most fields of engineering – is considered unacceptable [27]. Failures that arise with very low probability are unlikely to be revealed by testing. Cryptographic systems tend to have long lifetimes, because their use is often dictated by standards, and because certifying them is time-intensive. Thus, any testing done on a system will likely cover only a small fraction of the actual behaviors of the system during its deployed lifetime.

For the reasons just discussed, an appealing alternative to testing a cryptographic algorithm is to use formal verification techniques to produce a proof of correct operation for all possible inputs. Formal verification is the process of ensuring the correct behavior of computer systems through the application of rigorous analysis and mathematical reasoning. It has been a grand challenge in computer science for many years, but the dream of a highly-automated general-purpose formal verification system remains elusive. Formal verification is challenging for both theoretical reasons (many program properties, such as termination and program equivalence, are undecidable in general) and practical reasons (formal verification requires a detailed specification of correct behavior, and insight from a sophisticated human user is often required to guide a verification system to the correct proof).

One way to address the challenges of formal verification is to trade off generality for automation. This is what Axe does. Rather than attempt to be a general purpose software verification system, Axe is targeted to the verification of a restricted class of programs that includes cryptographic building blocks. This trade-off allows Axe to achieve a large degree of automation.

1.2 Overview of Axe

Axe is a system for performing equivalence checking; it can be used to prove that two representations of the same algorithm produce the same outputs, for all valid inputs. For example, Axe can prove that an optimized Java implementation of the AES block cipher produces exactly the same ciphertext as an executable formal specification of AES. It can also prove that two Java implementations of AES produce bit-for-bit equivalent outputs.

Axe deals with computations that are sequential (that lack parallelism) and terminating (that eventually produce outputs, rather than maintaining ongoing interactions with their environments, as do reactive systems). Cryptographic algorithms make up an important and challenging subclass of these computations, and Axe has been applied to verify block ciphers, stream ciphers, and cryptographic hash functions.

Like any equivalence checking tool, Axe can increase confidence in an implementation's correctness by proving it equivalent to a formal specification or a reference implementation. Ideally, several independently created implementations of the same algorithm would all be proven equivalent to the formal specification of the algorithm. In such a situation, either all the implementations (and the formal specification) are correct, or they all have the same bugs. In the case of the AES block cipher, for example, Axe has been applied to four implementations and has proven them all equivalent to a formal specification.

All of the cryptographic programs verified by Axe contain loops[1]. Loops provide a major challenge to automated verification, because verifying their properties traditionally requires the formulation of inductive loop invariants. When full functional correctness is the goal, these loop properties must be strong enough to capture the complete behavior of the loops (for example, Axe uses inductively-strong "connection predicates" to prove equivalence of two loops, as described in Section 12.1).

1.2.1 Complete Loop Unrolling

Axe employs two strategies when dealing with loops. First, when possible, Axe deals with loops by completely "unrolling" them[2]. Complete loop unrolling is only possible when the number of iterations of the loop is bounded. If the number of loop iterations is actually fixed (such as for the main encryption loop of AES, which always executes exactly 10 times when the key length is 128 bits), then the loop can be replaced with an equivalent loop-free term which contains the appropriate number of copies of the loop body, to be executed in sequence (e.g., 10 copies of the body of the AES encryption loop). If the number of loop repetitions is not fixed but is bounded, complete unrolling can still be used, by including if-then-else operations to govern how many copies of the loop body are actually executed. In this case, the bound on the number of iterations limits the total number of copies of the loop body that must be present in the unrolled representation.

[1] As described in Section 4.11, Axe's intermediate representation actually uses recursive functions to model loops. Throughout this thesis, any discussion of operations performed by Axe on "loops" will actually refer to the corresponding recursive "loop functions."

[2] Complete unrolling is not to be confused with partial or "constant factor" unrolling, described in Section 12.2.1

To apply the complete unrolling method to a loop whose iteration count is not fixed but is restricted to a few possibilities, one can simply perform several proofs. For example, to verify an implementation of AES encryption, which supports keys of 128, 192, or 256 bits (corresponding to 10, 12, or 14 iterations of the encryption loop), one would perform three Axe proofs.

Complete loop unrolling avoids the need to detect and prove loop invariants, but the terms arising from complete unrolling can be very large. As described in Section 4.8, they are especially unwieldy because they often have massive sharing of subterms. A major strength of Axe is that it can perform equivalence checking on such terms. This is enabled by Axe's use of a compact internal term representation that takes advantage of shared subterms. (Axe represents terms as directed acyclic graphs, or "DAGs," in which each subterm can be shared by multiple parents.)

Axe performs equivalence checking of large terms, such as the unrolled representations of two equivalent cryptographic programs, using a phased approach. It begins by building a "miter term" representing the equality of the terms being compared. The goal of equivalence checking is to reduce this equality to true. Axe first attempts to prove the miter through rewriting. That is, it repeatedly applies local, equivalence-preserving simplification rules. Axe does so first at the "word-level," in which operations deal with entire bit-vectors, (e.g., the exclusive-or of two 32-bit values). Then Axe performs "bit-blasting", in which word operations are replaced by equivalent concatenations of single-bit operations. Finally, Axe simplifies the bit-blasted representation. When rewriting, Axe also uses domain-specific rewrite rules that are designed to handle common coding idioms in cryptographic code.

If rewriting fails to prove the miter, Axe attempts to break down the proof into smaller proofs by discovering internal equivalences (pairs of subterms in the miter that appear to be equivalent as the miter is evaluated on many random inputs). Axe then performs "sweeping and merging," in which it attempts to prove these internal equivalences in a “bottom-up” fashion, starting with the smallest terms. When Axe is able to prove an equivalence, it “merges” the terms involved, replacing one of them with the other. This helps make the two “sides” of the miter more similar. If all goes, well, the sweeping process will end with the two computations having been merged together completely, making the miter trivially true. The proof justifying a merge is performed by either Axe’s own theorem prover or the external STP decision procedure [29] for bit-vectors and arrays. In order to make the STP queries tractable, Axe uses a sound heuristic to cut out irrelevant parts of the queries.

If sweeping and merging fails to prove the miter, Axe can perform a case split based on the condition of some if-then-else operator in the miter (if any). After splitting, Axe performs equivalence checking recursively for both cases.

Complete unrolling was used to verify several block cipher implementations provided by Sun Microsystems (AES, Blowfish, DES, Triple DES, and RC2) as well as several implementations from the open source Bouncy Castle Project [7] (Blowfish, DES, Triple DES, RC2, RC6, Skipjack, and three implementations of AES). These proofs are discussed in Section 2.1.

When an algorithm cannot be completely unrolled because the input size is not fixed, one can often perform a restricted proof for a particular input size. Specifying the input size usually bounds the number of loop iterations, and Axe can then apply complete unrolling to verify the implementation for that input size. For example, Axe has been used to perform complete unrolling proofs for implementations of the SHA-1 and MD5 hash functions, each on inputs of 32, 512, and 4096 bits. Of course, such proofs provide no guarantees of correct operation for the vast number of other possible input sizes.

1.2.2 Inductive Loop Proofs

Axe can also deal directly with non-unrollable loops, including the main loops of stream ciphers (Section 2.2) and cryptographic hash functions (Section 2.3). These algorithms have no useful limitations on their input sizes, and so their main loops, which must at least process all of the input, cannot be completely unrolled[3]. When non-unrollable loops are present, Axe has no choice but to attempt to reason about them inductively.

An Axe proof of a miter that contains loop functions has the same basic structure as the non-loop proofs described in the previous section; Axe simplifies the miter and sweeps upward, proving equivalences and merging pairs of equivalent nodes. Eventually, Axe will reach a pair of nodes that are supported by loop functions. In this case, Axe analyzes the behavior of the supporting loops. To do so, it evaluates the miter on random test cases, observes the executions of the loop functions, and makes conjectures about the loops' behavior. These conjectures may include properties of a single loop (traditional loop invariants and claims about the final values of the loop variables when the loop exits) and "connection" relationships between two loops (predicates that indicate how the variables of the two loops correspond). Loop connections are used to prove the functional equivalence of two loops.

Axe attempts to prove the conjectured loop properties by induction. The inductive step of each proof (which involves reasoning about a single execution of each loop body) is done by recursively calling the Axe Equivalence Checker. The equivalence

[3] Some algorithms technically do have limits on their input sizes. For example, SHA-1 requires its input to contain fewer than 2^{64} bits. Axe typically adds the restriction that each input be able to fit into an array in the Java programming language (and so must contain fewer than 2^{31} elements). However, neither of these restrictions is helpful in practice; even though the set of messages shorter than 2^{31} bytes is finite, there are too many possible message lengths to perform a complete unrolling proof for each one. Nor is it tractable to reason about a completely unrolled representation containing $2^{31} - 1$ copies of a loop body

checker handles any nested loops, usually by unrolling them. Such a "hybrid" approach, in which inner loops are completely unrolled, while non-unrollable outer loops are dealt with inductively, works well for cryptographic code. This is because cryptographic algorithms that support inputs of arbitrary size often process their inputs in fixed-size blocks. The computation performed on an individual block can usually be completely unrolled.

Sometimes the loops in two different implementations of an algorithm fail to correspond exactly. In such cases, Axe can perform transformations to help synchronize the loop structures of the programs being compared, so that loops correspond one-to-one between programs, with corresponding loops performing the same number of iterations. These transformations include "constant factor unrolling," loop splitting, "uncdring," putting loop functions into "simple" form, dropping redundant loop variables, and making "list builders" tail-recursive. Axe can also "streamify" the composition of a producer and a consumer (yielding a single equivalent loop that produces and then immediately consumes its data).

1.3 What Axe Proves

Axe proves that an implementation of a cryptographic algorithm always computes the result dictated by a formal specification, or that two implementations always produce identical results. The guarantee provided by Axe is that the outputs are bit-for-bit equivalent. This is a strong correctness property. (Weaker correctness properties for cryptographic code, such as the fact that encryption and decryption are inverses, are discussed in Section 3.7.)

Axe does not prove anything about the cryptographic strength of the underlying algorithms. Indeed, it is not clear how such a proof would be performed, and our approach is to leave such analysis to professional cryptographers[4]. The purpose of an Axe proof is to verify that the underlying algorithm was correctly implemented in a real system (e.g., a Java program).

Axe's notion of equivalence deals exclusively with the input-output behavior of computations, not with their resource usage (such as the amount of time, memory, or electrical power used). An optimized implementation may use fewer resources than an unoptimized version of the same algorithm, yet Axe would consider the two to be equivalent. Also, the equivalence-preserving transformations that Axe performs may change the resource usage of the computations involved. Because Axe does not consider resource usage, implementations verified by Axe may be susceptible to side-channel attacks such as timing attacks or power analysis.

Also, since Axe applies only to the functional (input-output) behavior of algorithms, an Axe proof about a Java class does not cover all possible interactions (sequences of method calls) with the class. As described in Section 9.5, Axe verifies the computation performed by calling the methods of the class in the usual order.

1.4 Challenges Addressed

Verification of cryptographic building blocks would be challenging without the support of a formal tool like Axe. In general, the difficulty of equivalence checking correlates with the degree of difference between the two implementations being compared. In many applications of equivalence checking (such as those discussed in Section 3.4), the implementations being compared are quite similar, because they represent successive refinements of the same design. In contrast, Axe usually deals with implementations created by different authors, for which the differences may be larger and harder to detect.

[4] Some of the algorithms dealt with by Axe, such as RC4 and MD5, are now considered somewhat insecure.

Many implementation differences are due to optimization. It is often necessary for cryptographic code to be fast, but an optimized implementation is more difficult to prove equivalent to an (unoptimized) specification. A common optimization is "constant factor" loop unrolling, in which each loop iteration performs more than one unit of work. Traditional inductive verification of such a program would have to deal with the unrolling, perhaps by first applying program transformations to synchronize the loops. When possible, Axe avoids the whole issue by completely unrolling loops. Otherwise, it performs synchronizing transformations.

Another common optimization is to pack data into machine words (e.g., packing 4 bytes into a 32-bit word). A typical "round" of computation would involve unpacking the data, operating on it in some way, and then packing the updated data back into machine words. Complete loop unrolling can eliminate the need to handle this issue. This is because the unrolled term will contain many instances of unpacking (done at the start of one round) composed with packing (done at the end of the previous round). Simplification rules can often simplify such a composition automatically.

Other optimizations may be present as well. For example, sequences of logical operations may be replaced by lookups into tables of previously computed results. However, cryptographic programs are often structured as sequences of "rounds," and optimizations rarely cross round boundaries. That is, results computed after round n of one implementation usually agree with the results after round n of another implementation of the same algorithm, even if the implementations of the rounds differ greatly. Axe exploits these correspondences by finding internal equivalences between unrolled cipher computations (as described in Section 8.3.2) and by heuristically cutting the proofs of these equivalences, so that irrelevant details of the computation of prior rounds are not passed to STP (as described in Section 8.3.3).

The disadvantage of completely unrolling cryptographic computations is that the resulting terms can be very large, due to massive sharing of subterms. Axe is built on top of the ACL2 theorem proving system [42], but ACL2's tree-based term language prevents it from efficiently representing and reasoning about such large computations (see Section 4.8). This deficiency was the initial impetus for the creation of Axe. Axe's intermediate representation, based on DAGs and used throughout the tool, allows it to deal efficiently with unrolled computations featuring massive sharing.

Cryptographic code often contains many bit-manipulating operations, such as the exclusive-or (XOR) operation. Because XOR is associative and commutative, implementations may differ in how they express the XOR of the same set of values, and these differences present a challenge to equivalence checking. Some implementations contain so many XOR operations that special handling of them by the Axe Rewriter is advantageous, as described in Section 5.10. Implementations may also differ on how they implement individual bit-vector operations, such as rotations; Section 6.1 describes domain-specific rewrite rules that Axe uses in such cases.

Some cryptographic algorithms contain multiplication operations. For example, the RC6 block cipher [56] repeatedly performs multiplication on 32-bit words. Since tools based on boolean satisfiability (SAT), such as the STP tool used by Axe, give poor performance when applied to goals that include multiplication, Axe performs aggressive rewriting of the miter before attempting sweeping and merging. In the case of RC6, rewriting suffices to prove the equality of two implementations in a matter of a few seconds.

A final challenge is that Axe deals with programs written in a real-world language (Java) that was not created to be especially amenable to formal verification. Chapter 9 describes how Axe extracts the computation performed by a Java program. However, even after transformation to Axe's intermediate representation, vestiges of the Java code remain. For example, arithmetic operations are usually performed on 32-bit signed integers, and verification must deal with the possibility of arithmetic overflow. Axe deals with this in two ways: by using rewrite rules that apply only when overflow can be proven not to occur, and by relying on STP's bit-accurate handling of arithmetic operations.

Using the techniques mentioned above and described in detail in the rest of this thesis, Axe provides considerable automation to the process of formally verifying implementations of cryptographic building blocks.

1.5 System Architecture

The Axe system has several main components: a rewriter (Chapter 5), a large library of proven simplification rules, including several key domain-specific rewrites (Chapter 6), a theorem prover (Chapter 7), an equivalence checker (Chapter 8), including support for transforming and reasoning about recursive functions (Chapter 10), and a loop property detector (Chapter 13). Axe also includes a library of formal specifications for many cryptographic algorithms (Section 4.12), and several tools for dealing with Java bytecode programs (Chapter 9), including a formal model of the Java Virtual Machine, a Java class file parser, and a decompiler that turns Java bytecode programs into equivalent mathematical terms in Axe's intermediate representation.

Axe is built on top of the ACL2 theorem proving system. ACL2 is both a programming language (essentially a side-effect-free subset of Common Lisp [33]) and an industrial-strength theorem prover for programs expressed in that language. The main Axe system consists of tens of thousands of lines of ACL2 code. Axe handles most of the proof work itself but calls ACL2 to assemble the pieces of each induction proof. In addition, Axe contains hundreds of rewrite rules, almost all of which have been proven as ACL2 theorems. Axe also relies on the STP decision procedure for bit-vectors and arrays. STP in turn relies on a SAT solver (Minisat [50]).

Chapter 2

Cryptographic Algorithms Verified

Axe has been used to verify several kinds of cryptographic building blocks, including block ciphers, stream ciphers, and cryptographic hash functions. These algorithms all eventually terminate and so can be modeled as mathematical functions. Of course, the functional representations are very complicated, as might be expected for algorithms whose purpose is to obscure information. This chapter describes the algorithms to which Axe has been applied and the proofs performed. The Java code checked by Axe comes from two sources: the open source Bouncy Castle project (under `org.bouncycastle.crypto`) and Sun Microsystems (under `com.sun.crypto.provider`).

2.1 Block Ciphers

A block cipher is a cryptographic algorithm that encrypts and decrypts small, fixed-size messages using a shared, secret key. A typical block cipher supports two operations: encryption of the input "plaintext" using the key to produce an indecipherable "ciphertext," and decryption of the ciphertext, using the same key, to recover the original plaintext. A good block cipher reveals very little information to anyone who lacks the key. Many block ciphers support several key lengths, with longer keys providing more security.

One important block cipher is AES (the Advanced Encryption Standard [5]). AES takes as input a plaintext of 128 bits and a key of 128, 192, or 256 bits. Its output is a 128-bit ciphertext. (In a block cipher, the plaintext and ciphertext are always the same size, the "block size" of the cipher.) Decryption takes the ciphertext and the key and returns the original plaintext. For even the smallest key size (128 bits), AES has 2^{128} possible keys and 2^{128} possible plaintexts, for a total of 2^{256} possible inputs. This is too many inputs to test exhaustively. Thus, a formal proof, such as the one performed by Axe, is required to verify the correctness of an implementation.

For the AES block cipher, Axe has verified four implementations. These include Sun's implementation (class `com.sun.crypto.provider.AESCrypt`) as well as the "light," "regular," and "fast" implementations from Bouncy Castle (classes `AESLightEngine`, `AESEngine`, and `AESFastEngine` under `org.bouncycastle.crypto.engines`). The Bouncy Castle implementations differ in the amount of optimization performed, but even the "light" version contains optimizations. For each implementation, proofs were done for both the encryption and the decryption operations for each of the three AES key sizes (128 bits, 192 bits, and 256 bits). Thus, Axe was used for a total of 24 proofs about AES (4 implementations times 2 operations times 3 key sizes).

2.1.1 Example Proof: AESFastEngine

The most heavily optimized of the Bouncy Castle AES implementations is `AESFastEngine`. For speed, it replaces certain sequences of logical operations with lookups into pre-computed tables of values (eight tables, each containing 256 32-bit words). Also, the cipher state, a four-by-four array of bytes, is packed into 32-bit machine words (by columns), and the encryption and decryption loops are partially unrolled, performing two rounds of work per loop iteration. Six proofs were performed to verify `AESFastEngine` (to cover encryption and decryption for each key size). For each proofs, the loops in the specification and implementation can be completely unrolled. For example, the main encryption loop performs 10 iterations when a key of 128 bits is used.

For AES-128 encryption, the unrolled implementation of `AESFastEngine` contains 1990 nodes (without bit-blasting) and is generated in about 12 seconds. The corresponding unrolled version of the formal specification contains 2729 nodes. It contains more nodes than the implementation because the sequences of operations in the implementation have been replaced by table lookups[1]. Unrolling the specification takes about 1 second. To do the equivalence proof, Axe forms a miter for the equality of the specification and the implementation. It then simplifies and normalizes the miter, including applying bit-blasting. The resulting miter has 20611 nodes. Axe then performs sweeping and merging, yielding 3684 sets of probably-equal nodes, which it proves by making 1152 calls to STP. (Some pairs of nodes can be trivially merged without calling STP due to prior merging having made their argument lists identical.) After 83 seconds, Axe merges the top nodes of the specification and implementation, thus finishing the proof. The process is completely automatic; no code annotations (e.g., loop invariants) are required.

2.1.2 Example Proof: RC6

Axe has also been used to verify the RC6 block cipher [56] (class `org.bouncycastle.crypto.engines.RC6Engine`). RC6 can be challenging to verify because it contains multiplication operations on 32-bit operands, and multiplication is known to be problematic for SAT-based tools. Normally the sweeping and merging phase of the Axe Equivalence Checker makes many calls to STP, which ultimately calls SAT. However,

[1] A large lookup table is represented in Axe as a constant array, and it is considered to have the same size as any other constant.

the Axe proofs about RC6 are done completely by rewriting the miters. Rewriting is always done before sweeping and merging, and in this case (and several others) it suffices to prove the entire miter. The proof is quite fast: Unrolling the formal specification takes less than a second, symbolically executing the implementation (including applying many simplifications) takes about 10 seconds, and the equivalence proof takes less than 1 second.

RC6 makes use of bit-vector rotation operations, including "variable rotation" operations in which the rotation amount is not constant but depends on the inputs to the cipher. As described in Section 6.1, variable rotations provide a challenge to verification, and Axe contains rewrite rules to handle them appropriately. Such rewrite rules are critical to Axe's ability to rewrite the RC6 miter equality all the way to true.

Axe's proofs for the RC6 block cipher cover both encryption and decryption for the three supported key sizes (128 bits, 192 bits, and 256 bits).

2.1.3 Example Proof: Skipjack

Early block cipher verifications performed by Axe were done in parallel with the development of Axe itself. Thus, it was difficult to estimate the amount of work required to verify a cipher. Once the core functionality of Axe was complete, we conducted an experiment. We verified a new cipher, called Skipjack [48], from scratch, and timed how long it took. The entire process took one person less than three hours. This included time to get familiar with Skipjack, write a formal specification for it, debug and validate the formal specification using test vectors, and apply Axe to prove equivalence of the specification and a Java implementation (class `org.bouncycastle.crypto.engines.SkipjackEngine`). The computer time needed to replay the proof (unrolling the computations and performing the equivalence proof) is less than 15 seconds. The proof is done by rewriting alone.

2.1.4 Other Block Ciphers

Axe has been used to verify several other block ciphers. These include implementations of DES [21] from both Sun and the Bouncy Castle project (with proofs about both encryption and decryption for each) as well as the related cipher Triple DES (also for two implementations, verifying encryption and decryption for each). For Triple DES, equivalence proofs were also performed between the Sun and Bouncy Castle implementations, thus showing that Java implementations can be compared to each other directly, without the need for a formal specification. Proofs for DES and Triple DES were done by rewriting followed by sweeping and merging. (Rewriting alone was insufficient for these proofs, due to incomplete normalization of expressions involving lookup tables.)

Axe was used to verify encryption and decryption of the RC2 block cipher. Implementations from Sun and Bouncy Castle were proved equivalent to each other and to a formal specification (all proofs used keys of 64 bits, the default size). The proofs were done by rewriting alone.

Axe was also used to verify two implementations of the Blowfish block cipher [12]. Blowfish has a very large unrolled representation, due to the work done to set up the algorithm's "P-array" and "S-boxes" using the user's key[2]. Even when represented using Axe's compact intermediate representation, the expressions for Blowfish contain over 200,000 nodes (without bit-blasting). The Blowfish proofs were performed using rewriting alone, and each takes several hours to run. Two implementations (`com.sun.crypto.provider.BlowfishCrypt` and `org.bouncycastle.crypto.engines.BlowfishEngine`) were checked, for both encryption and decryption. Proofs were performed for keys of 64 bits and 448 bits (448 bits is the largest supported key

[2] The core encryption operation of Blowfish is a single execution of its "Feistel network," but the preceding process of setting up the requisite arrays involves several hundred executions of this same Feistel network!

size).

2.2 Stream Ciphers

A stream cipher is a cryptographic algorithm that, like a block cipher, encrypts and decrypts a message using a shared, secret key. Unlike a block cipher, a stream cipher can operate on a plaintext of any length. A stream cipher works by using the key to create a pseudo-random “key stream” which has the same length as the plaintext. Encryption involves simply XORing each byte of the plaintext with the corresponding byte of the the key stream. Decryption is the same operation as encryption; the same key stream is generated and XORed with the ciphertext, cancelling out the XOR operation performed during encryption.

Because a stream cipher operates on a message of unknown length, it cannot be verified by completely unrolling all of its loops. (There must be at least one loop that processes all the data, and the number of iterations of this loop will not have any small bound.) Thus, to verify the application of a stream cipher to an arbitrary message, Axe must discover and prove inductive loop properties. However, a more restricted proof can be obtained for a stream cipher by fixing the sizes of its inputs, then performing complete unrolling (since all loop iteration counts will be bounded) and applying the Axe Equivalence Checker.

Compared to other algorithms that operate on messages of unknown length (such as the cryptographic hash functions discussed in the next section), stream ciphers are easier to verify in one respect: they perform no padding of the input message. However, verification of stream ciphers may be complicated by the fact that their implementations may be “streamified;” instead of creating the entire key stream first and then XORing it with the message, an implementation can generate the bytes of the key stream “just in time” to combine them with the message bytes, never actually holding the entire key stream in memory at once.

Axe has been applied to verify a Java implementation of the RC4 stream cipher (class `org.bouncycastle.crypto.engines.RC4Engine`), as described in the next two sections.

2.2.1 Verifying RC4 for fixed-size inputs

Two Axe proofs were performed for RC4 on fixed-size inputs. First, the sizes of both the input message and the key were fixed at 64 bits. This allows complete loop unrolling to be performed. The unrolled specification contains 66420 nodes (without bit-blasting), and applying complete unrolling to the Java implementation (via symbolic execution) yields a representation with the same number of nodes. Axe can rewrite the equality of the two unrolled representations to true, and the entire process (unrolling both computations and proving them equal) takes less than one minute.

A proof for larger inputs was also performed. Axe verified the application of RC4 to a key of 2048 bits (the largest meaningful key size for RC4[3]) and a message of 4096 bits. For these larger inputs, the unrolled DAGs are larger (202496 nodes for both the specification and implementation). Again, the equality rewrites to true, and the entire process takes less than five minutes.

These proofs of RC4 for particular input sizes require no program annotations, but they do not guarantee correctness of RC4 for any of the vast number of other possible input sizes.

[3] Extra data beyond the first 2048 key bytes is ignored.

2.2.2 Verifying RC4 for all inputs

Axe has also been used to verify RC4 for all possible input messages whose length is less than 2^{31} bytes[4]. The proof covers all possible keys containing least one byte[5] and less than 2^{31} bytes[6]. The proof handles the loops without unrolling, but no program annotations are required from the user; Axe detects the necessary loop invariants and proves them by induction.

The DAGs involved in the proof of RC4 are much smaller than for a complete unrolling proof, but they contain recursive functions that model the loops, and the bulk of the computation is represented in the bodies of the loop functions. The specification DAG for RC4 contains only 6 nodes[7], but it includes three nodes that call recursive loop functions: `set-up-s-box` mixes the key data into the cipher's "s-box," `stream-bytes` creates the pseudo-random stream, and `bvxor-list` combines the stream with the input message by XORing corresponding bytes. (There is technically a fourth loop, which pre-initializes the s-box to contain the numbers 0 through 255, in order, before the key data is mixed into the s-box. Because that computation does not depend on the inputs to the cipher, it is manifested as a function applied to constant arguments. Axe simply evaluates the function call, yielding the constant array that contains the numbers 0 through 255. This eliminates the need to reason further about that loop.)

The DAG for the Java implementation is generated by decompiling the bytecode. (Because the number of loop iterations is not fixed, symbolic execution alone cannot

[4] This limit comes from the requirement that the message be able to fit into an array in the Java language, which limits arrays to $2^{31} - 1$ elements.

[5] RC4 does not allow 0-byte keys, presumably because the key length is used as the modulus of a *mod* operation.

[6] Although very long keys are allowed, RC4 only actually uses at most 256 bytes of any key.

[7] It results from simply opening up the non-recursive top-level function, `rc4-encrypt`, of the formal specification.

extract a representation of the computation performed by the code. The Java decompiler is described in Section 9.4.) The implementation contains three loops. The first pre-initializes the s-box to contain the numbers 0 through 255, in order. As for the corresponding loop in the specification, this loop is simply executed, producing the constant array containing the numbers 0 through 255.

The second loop, represented by the function `rc4-loop-2` (generated and named by the decompiler), mixes the key data into the s-box and is analogous to the specification function `set-up-s-box`. Axe is able to automatically prove that the two functions are equivalent. Before proving equivalence, Axe transforms `rc4-loop-2` slightly, to drop some loop variables that can be expressed in terms of others. These redundant loop parameters are detected by the observation of actual execution traces of the loop function. The most interesting is the variable `i1`. In the loop in question, index `i` increases by 1 each iteration, and index `i1` increases by 1 modulo the length of the key. Axe notices that the variables are related and in fact considers `i1` to be redundant, since it can always be calculated from `i` and the key length (which does not change during the loop). In particular, Axe proves by induction that `i1` is always equal to `(bvmod 31 i (len key))`. Axe then generates a new function which lacks the `i1` parameter and proves the equivalence of a call to the old function and a suitable call to the new function. The new function matches up better with the corresponding specification function, which lacks a variable that is incremented modulo the key length. (Like the transformed function, the specification function has a single variable that always increases, and a modulo operation is performed on it right before it is used.) The difference in loop indices would have to be dealt with by any verification method used to prove the implementation equivalent to the specification.

After the transformation, Axe is able to automatically prove the equivalence of the two loop functions. Based on test cases, it identifies loop variables that correspond between the two implementations, and it proves these facts by induction. The resulting rewrite rule is then used to prove the equality of the two probably-equal nodes supported by the loop functions.

The third loop in the implementation, `rc4-loop3`, actually corresponds to two loops in the specification. Instead of creating the entire pseudo-random stream and then combining it with the message, the implementation creates the stream bytes "just in time," never actually holding the entire stream in memory at one time. This is done by the single loop `rc4-loop3`. This streamification represents a major difference from the formal specification, in which two loops perform the same work; in the specification, `stream-bytes` creates the entire stream and `bvxor-list` combines the bytes of the stream with the corresponding message bytes. Since the loops in question cannot be unrolled, Axe must somehow handle the difference in loop structures between the two programs. It does so by transforming the specification into a "streamified" form. Based on analysis of the functions involved in the specification, Axe determines that a producer/consumer relationship exists; `stream-bytes` produces a list of values which `bvxor-list` then consumes in order (and `bvxor-list` keeps no history of previously consumed values). Axe then automatically generates a new function that represents the composition of `stream-bytes` and `bvxor-list` and generates a rewrite rule to put in the new function[8]. After streamification, Axe is able to prove the equivalence of the new composition function and the implementation function `rc4-loop3`, by finding and proving correspondences between the loop variables in the usual way.

The verification of RC4 for all inputs deals with loops inductively but still requires no program annotations from the user. The proof takes about 17 minutes of computer time.

[8] This transformation has been proven correct in the general case, but, unlike other transformations performed by Axe, each individual application of the transformation is not proved correct. For higher assurance, Axe could be improved to check each application of the streamifying transformation.

2.3 Cryptographic Hash Functions

A cryptographic hash function is an algorithm that takes a single message and generates from it a message digest (or signature). Usually the message can be of any length (possibly up to some large limit), and the size of the digest is fixed and relatively small. For example, the SHA-1 hash function [57] takes a message of less than 2^{64} bits and computes a 20-byte digest. Since there is no useful limit on the input size, verification of cryptographic hash functions cannot be performed by completely unrolling all the loops. Thus, as is the case for stream ciphers, an Axe proof for all input sizes involves discovering and proving inductive loop properties.

Cryptographic hash functions typically operate by dividing their inputs into fixed-size blocks (512 bits for SHA-1), with padding performed to deal with partially-filled final blocks. To pad, the hash function extends its input message with extra data to make its length a multiple of the block size. For cryptographic reasons, the padding operation is usually a bit complicated[9]. Padding can complicate verification if the formal specification and the implementation differ on when it is performed. In particular, the specification for a cryptographic hash function usually involves two basic steps: first the message is padded, and then the padded message is processed block by block. However, implementations often avoid representing the entire padded message in memory. Instead, they may first process all of the complete blocks and then, in a separate operation, handle the partially-filled final block (if any) and deal with padding. This difference arises in Axe's proofs of Java implementations of the SHA-1 and MD5 [55] hash functions (classes `org.bouncycastle.crypto.digests.SHA1Digest` and `org.bouncycastle.crypto.digests.MD5Digest`). It is somewhat akin to the issue of "streamification" of a stream cipher discussed previously.

[9] For SHA-1, the padding involves adding a single one bit, adding some number of zero bits, and then appending a 64-bit quantity representing the length of the original message. This process fills up the final 512-bit block of the message and possibly adds one additional block. For MD5 the padding is the same, except for the order of the constituent words of the 64-bit message length.

Because hash functions process their inputs in fixed size blocks, verifying their loops inductively is usually only necessary for the outer loops; any inner loops (which deal only with a single arbitrary block) can often be completely unrolled. This avoids the need to formulate loop invariants or perform loop transformations for any inner loops. Since the verification of hash functions' outer loops is challenging enough, such a hybrid approach is usually used by Axe.

Verification of cryptographic hash functions is complicated by issues of data layout and packing. For MD5 and SHA-1, the formal specifications deal with inputs that are sequences of bits. However, in the Java code, the inputs are sequences of bytes[10]. The correctness claims for the hash functions reflect this difference. For example, the correctness claim about the Java code for MD5 says that, for any sequence of bytes, converting the bytes to a sequence of bits and then calling the specification function yields the same hash value as calling the Java implementation on the original sequence of bytes.

Several other levels of data packing are involved in the verification of SHA-1 and MD5. First, bytes are packed by groups of 4 into 32-bit words. (SHA-1 and MD5 differ on the endianness of this packing.) Then words are grouped in groups of 16 to make the message "blocks." The input message as a whole thus can be viewed as containing 0 or more complete blocks and perhaps one partially-filled final block. As is the case for padding, data packing can cause differences between the specification of a hash function (which may pack the whole padded message before processing the blocks) and an implementation (which may perform the packing only when the relevant data is actually processed).

[10] The Java implementations thus do not support inputs with partially filled bytes. Throughout this dissertation we only consider inputs for which the number of bits is a multiple of 8.

2.4 Verifying MD5 for fixed-size messages

As is the case for stream ciphers, cryptographic hash functions can be verified using the method of complete unrolling, if one is willing to fix the lengths of the inputs and thus obtain weaker statements of correctness. Axe has been used to formally verify both the SHA-1 and MD5 hash functions on messages of 32 bits, 512 bits[11], and 4096 bits. The equivalence proofs for 4096-bit inputs take 282 and 873 seconds for MD5 and SHA-1, respectively.

2.5 Verifying MD5 for all inputs

Axe has also been used to verify the Bouncy Castle implementation of MD5 for all message lengths less than 2^{31} bytes[12]. The proof is quite challenging, due to differences between the specification and implementation. Unlike the proofs described so far in this thesis, some program annotations are required from the user. (It may be possible to improve Axe to discover the necessary annotations automatically by examining execution traces.)

The heart of the verification of MD5 consists of dealing with two loops, `md5-loop-5` from the Java code, and the `process-blocks` function from the formal specification. These loops differ in the amount of work they perform per loop iteration; `process-blocks` handles an entire block per iteration, but `md5-loop-5` handles only a single word. `md5-loop-5` copies words into a buffer, and, after every sixteenth word (when the buffer is full), it processes the complete block contained in the buffer (by calling a complicated routine to update the MD5 hash). In some sense, `md5-loop-5` actually represents two loops, an outer loop which processes the

[11] After padding, a 512-bit message will always occupy 1024 bits; thus hashing such a message will involve processing two 512-bit blocks.

[12] As mentioned above, the input must fit into a Java array, and messages with partially-filled bytes are not supported.

blocks and an inner lop which processes the words within a block. In order to prove the equivalence of the specification and implementation, `md5-loop-5` must be unrolled by a factor of 16, because 16 of its iterations correspond to a single iteration of `process-blocks`. Unrolling by 16 produces a new loop for which each iteration processes an entire block.

Even after this constant factor unrolling, the loops do not quite match up, because `process-blocks` processes the message after padding is performed, while `md5-loop-5` processes only the complete blocks of the input message. Any final, partially filled block is handled by other code, and padding is also dealt with outside `md5-loop-5`[13]. Because the specification loop handles more blocks of data than the corresponding loop in the Java code, we split the specification loop using Axe's loop splitting transformation. In particular, we split off the processing of the complete blocks, which corresponds to the first `(bvdiv '32 (len input) '64)` loop iterations (each block contains 64 bytes). The extra iterations of the specification loop (there may be one or two of them, depending on whether padding causes the message to grow by an entire block) can be completely unrolled.

The implementation and specification also differ in how data is packed into blocks. In the specification, each block is a list of 16 32-bit words, but the implementation operates on a raw, unstructured array of bytes. The key annotation is:

```
(prefixp (group '16 (map-packbv '4 '8 (map-reverse (group '4 (nth '5 fparams)))))
         gblocks)
```

Here `fparams` is a tuple representing the parameters to (the unrolled version of) `md5-loop-5` and `(nth '5 fparams)` is the input byte array. The annotation says that

[13] In fact, the situation is even more complicated, because the Java loop, which conceptually processes "complete words," and which after unrolling thus conceptually processes "complete blocks," will actually leave the last complete block unprocessed if the message length is an exact multiple of the block size. We don't believe this is a bug, due to how the padding is done later. However, to simplify the program annotations required for Axe, we changed a `>` sign in the code to a `>=` sign. We would like to return to this issue and verify the unmodified code, perhaps after Axe is improved to generate the correct annotations automatically.

if you group these bytes into groups of 4, reverse the bytes within each group (this reflects the endianness of the word packing), pack each group of 4 bytes into a word, and then group the words into blocks of 16 words, the result is a prefix of `gblocks`. The variable `gblocks` represents the entire input message, after padding, packed and grouped into blocks; it is one of the loop variables of `process-bytes`. The function `group` discards partially filled groups; thus, for the Java code, the annotation only deals with complete blocks of the input. The use of `prefixp` is necessary because `gblocks` contains extra data from the padding operation and from any partially-filled final block. Axe contains an extensive set of rewrite rules for dealing with the functions `prefixp`, `group`, `map-packbv`, and `map-reverse`. (The latter two are defined in terms of `packbv` and `reverse`, about which Axe also has rewrite rules.) Given the annotation above, and a few other less interesting ones, Axe is able to prove the equivalence of the unrolled version of `md5-loop-5` and `process-bytes`. The loop reasoning is the hard part of the proof of MD5. All of the remaining reasoning, including the details of processing an arbitrary block and the details of the padding done at the end of the message, can be expressed without loops. Some of the loop-free computations are, however, quite complicated due to if-then-else expressions (arising especially from the "residual computation" left after unrolling `md5-loop-5` by a factor 16). The if-then-else operations on the implementation side of the miter have no corresponding operations on the specification side and thus prevent probably-equal nodes from being identified, for technical reasons[14], until Axe splits the miter into cases. In all, 64 cases arise, corresponding to the different possibilities for the low order 6 bits of the message length, which represent the number of bytes in the final partially-filled block. The full proof takes several days to run[15] The running time

[14] Briefly, when Axe evaluates an in-then-else node, it evaluates the test and then only evaluates the relevant branch. So not all nodes are "used" on every test case, and nodes are only considered probably-equal if they are used on the same set of test cases.

[15] During the development of the proof we shortened the debug cycle by arbitrarily choosing a value for the low order 6 bits of the message length. Most of the annotations are the same for the

of the proof could be improved by attacking the cases in parallel, by addressing the issues that prevent identification of probably-equal nodes when if-then-else operations are present, or by making the implementation of Axe more efficient.

While proof of the MD5 hash function for all message lengths requires much more time than the proof for a restricted message length (and requires some user annotations), it provides a much stronger result. Axe was also used to perform a similar verification of the SHA-1 hash function for all inputs.

restricted case and for the full proof.

Chapter 3

Related Work

A major strength of Axe is that it combines techniques from several research areas. Axe can be seen as an extension of combinational equivalence checking that supports arbitrary operators in miters (including word-level operators and even loop functions that it reasons about using induction). Or it can be seen as system for performing inductive verification that can also sometimes simply unroll loops (including inner loops), representing the results as DAGs and using simulation and sweeping and merging to perform the proofs. This chapter describes research areas that are related to Axe and indicates how Axe fits into the larger research picture.

3.1 Combinational Equivalence Checking

Combinational equivalence checking (CEC) is a very successful technique in the field of hardware verification, and Axe borrows several key ideas from it when dealing with unrolled loops. A combinational circuit is one which includes no feedback loops or state holding elements[1], and combinational equivalence checking is the process of

[1] Despite not dealing directly with state-holding elements, such as registers, CEC can be a key component in reasoning about stateful systems using the techniques of bounded model checking or sequential equivalence checking.

proving that two such circuits with the same inputs always have the same outputs. This is usually done by combining the circuits into a "miter circuit" [14], whose output will be 1 on any input for which the original circuits differ, and then proving that the miter output is 0 for all inputs. Such a miter is very similar to the miters built by Axe, with a few major differences[2]. First, Axe's miters can contain any operators, whereas the standard in combinational equivalence checking is to use the "And-Inverter Graph" [6] [44], a simple, compact representation using only and-gates connected by possibly inverted edges. Axe's more flexible representation allows it to apply transformations at the word level (such as commutativity of 32-bit arithmetic operations), before bit-blasting the DAG into single-bit operations. Also, because DAG operators can be recursive functions that model loops, Axe provides a unified framework for applying combinational equivalence checking techniques and inductive loop verification

Miters in CEC tools are typically represented as DAGs that are built using "strashing," or structural hashing, in which no two nodes are allowed to be identical; Axe follows the same policy, and it prevents exponential explosion in the terms size when representing unrolled cryptographic loops. The process of proving a CEC miter usually begins with random simulation (i.e., evaluation of the nodes of the miter on random test cases) to discover internal equivalences. The use of simulation in this way goes back at least to Berman and Trevillyan's work in 1989 [10]. Their approach, as does Kuehlmann's [37], involves proving the discovered equivalences in a bottom-up fashion, much like Axe does in its sweeping and merging phase. Early approaches used binary decision diagrams (BDDs [15]) during sweeping and merging, but newer work – including later work done by Kuehlmann and others [63], and

[2] A minor difference is that Axe's miters return true, not 0, on inputs for which the implementations agree.

the checker ABC [44] – involves "SAT-sweeping"[3] (or the related technique of "fraiging"[4]), in which functionally equivalent nodes are proved equivalent using a boolean satisfiability solver.

A major difference between Axe and other combinational equivalence checkers is in how two nodes are proved equivalent to justify merging them. Instead of building BDDs or calling SAT directly, Axe calls the STP prover, and it does so using cutting heuristics to remove irrelevant information from the goals. This heuristic cutting seems key to the success of Axe; without it, STP bogs down on large, unrolled cryptographic computations.

An experiment attempting to replace Axe's equivalence checking phase with a simple call to an off-the-shelf equivalence checker (ABC) was not successful. We used Axe to unroll all of the loops in the Java code and the formal specifications of several ciphers but refrained from applying Axe's word level transformations, so that we could test their effects. (Some transformations were necessary, e.g., bit-blasting and getting rid of array operations, to convert the DAGs to use only operators supported by ABC.) For "light" (not heavily optimized) AES, ABC took about five times as long as Axe (385 seconds vs. 72 seconds). For "regular" AES, ABC took about 10 times as long (2149 seconds vs. 196 seconds). For RC6, a more challenging example due to the presence of "variable rotations" and 32-bit multiplications, ABC crashed (with a bus error) after about 1.5 days. The proof in Axe takes about 12 seconds. This experiment seems to indicate that rewriting, such as that performed by Axe, is

[3] Some systems use both BDDs and SAT [38].

[4] "Fraiging" (whose name comes from "functionally-reduced AIG"), can be seen as a semantic analogue of strashing. Whereas strashing never builds two nodes that are syntactically identical, fraiging seeks to avoid building two nodes that are functionally equivalent. Thus, when building a new node, one either uses simulation to find inputs that distinguish it from all preexisting nodes, or one proves it equivalent (usually by calling SAT) to some already existing node. Equivalence checking then reduces to the question of whether the top nodes of the two implementations are assigned the same node during fraiging. Of course, deciding whether two nodes are equivalent can be very hard, and timeouts are usually used in practice.

needed to verify a wide range of unrolled cryptographic programs.

Non-unrollable computations (such as stream ciphers and cryptographic hash functions) fall outside the scope of traditional CEC tools, because such tools require fixed input sizes and cannot deal with loops using inductive methods. When loop functions are involved, Axe still follows the basic sweeping and merging approach, but uses the Axe Prover (Chapter 7) to prove pairs of nodes equivalent. The Axe Prover can make use of lemmas that characterize loop functions, and such lemmas are generated and proved by Axe as needed (guided by analysis of execution traces). Also, when loop functions are present, proofs of merges are done using approximate contextual information provided by if-then-else nodes, as described in Section 10.5.1. Traditional CEC methods don't seem to use this sort of contextual information, perhaps because AIGs do not include if-then-else operators.

Axe performs extensive rewriting at the word level (using general purpose rules and rules that handle several domain-specific coding idioms in cryptographic code) before attempting sweeping and merging on a miter. Rewriting is sufficient by itself to complete the proofs of the miters for several cryptographic implementations. Some CEC approaches use a form of rewriting, but it is usually at the relatively low level of AIGs. For example, techniques from logic synthesis can be used to try to increase the sharing in the miter, which usually speeds up the equivalence check [45].

There are some techniques used by modern CEC tools that Axe does not use (but could be improved to use). These include interleaving sweeping attempts (subject to timeouts) with attempts to call SAT on the entire miter (which can sometimes prove the miter even without merging all equivalent node pairs), identifying node pairs that are complements of each other (Axe only finds equalities), and performing intelligent simulation in which interesting patterns (those that disambiguate nodes) are used to generate additional interesting patterns (by flipping every possible bit, one at a time) [46].

3.2 Cryptol Verification

Perhaps the work most closely related to Axe's complete unrolling proofs is that being done by Galois Connections, Sean Weaver, and the National Security Agency in verifying cryptographic implementations against specifications written in the Cryptol [17] language. Cryptol is a Haskell-based domain-specific language for cryptography. Cryptol specifications are designed for clarity and are analogous to the formal specifications used by Axe.

Galois has implemented a verifying compiler for Cryptol [52]. In contrast to a verified compiler, whose operation has been proved correct once-and-for-all (which can be very difficult), a verifying compiler performs a proof each time it is run (to ensure that the compiler's output correctly corresponds to its input). The Cryptol tool chain includes significant support for equivalence checking [23] [24]. In particular, it can generate AIGs from Cryptol programs, which can then be compared with the built-in jaig equivalence checker (which in turn calls the Minisat SAT solver [50] and uses some Cryptol-specific heuristics). Cryptol programs can also be compared with AIGs generated from C programs or VHDL descriptions.

The Galois tools' main equivalence checking approach is quite similar to how Axe verifies programs by complete unrolling. First, the computations checked must be finite (unrollable), and all data sizes must be fixed[5]. All loops are unrolled through symbolic execution, and the result is represented as a DAG, with sharing of subexpressions. As in Axe, simulation is used to find pairs of equivalent nodes, and nodes are merged in a bottom-up sweep. The Galois system can quickly perform equivalence checks on implementations of AES or DES encryption (taking 30 seconds to a few minutes). However, some proofs take longer (17.5 hours for a proof of the

[5] Equivalence checking deals with "monomorphic" programs, whereas Cryptol specifications in general are often polymorphic, because they support multiple data sizes.

Skein hash function [26] with the input size fixed to 256 bits[6], and 10 minutes to check the application of MD5 on a 2-bit input). Unlike Axe, the Galois equivalence checking system does not seem able to handle implementations of the RC6 cipher. (Axe proves RC6 via rewriting.) Also, it does not seem to support inductive reasoning of the sort Axe uses to verify non-unrollable computations.

3.3 Inductive Assertions and Cutpoints

A very well established technique in program verification is the use of "inductive assertions." to label the "cutpoints" of a program [30] [28]. Given the control flow graph of a program, one designates certain program locations to be "cutpoints." Usually these include the entry and exit points of the program and enough intermediate points to "cut" all of the loops (so that any execution path between two cutpoints has bounded length). One then annotates the cutpoints with assertions about the program variables. This must be done in such a way that the assertions are true (holding whenever control reaches the corresponding cutpoints) and "inductive," meaning that, for any simple path between two cutpoints, if the assertion for the cutpoint at the start of the path holds initially, then, when control reaches the end cutpoint (having executed the statements along the path), the assertion of the ending cutpoint holds. An inductive argument (on the number of such execution steps performed) can then be used to establish the partial correctness of the program (i.e., that if the input to the program satisfies its preconditions, and the program terminates, then the output satisfies the postconditions). Each simple path corresponds to a verification condition, generated according to the semantics of the language statements along the path (e.g., by propagating the assertion backward over the code using the method of weakest preconditions [13], or by symbolically executing the code in the forward

[6] using ABC to do the equivalence checking

direction [43]) that must be discharged. Proving the verification conditions is done using a theorem prover or a decision procedure and usually does not involve induction (since no loops occur along a simple path).

Axe can also prove properties of programs inductively (e.g., proving that a loop function preserves its invariant), but the method used is somewhat different. Instead of control flow graphs, Axe is based on DAGs (representing mathematical terms) that contain recursive functions to model loops. Of course, the process of reasoning about a loop does in both cases boil down to formulating and proving an inductive loop invariant. However, Axe's method of handling nested loops is more hierarchical; to reason about a DAG requires performing inductive proofs about the loops it contains, and the body of each loop may in turn be represented as a DAG, which may contain nested loops, and so on. Axe's reasoning is in some sense "inside-out," with the proof about the innermost nested loop preceding the proof about the next loop outward in the hierarchy, and so on. In contrast, a cutpoint proof deals with nested loops in a less structured way; the program is simply a collection of simple paths, and each path has to be checked (to assure that it preserves annotations). Care does have to be taken to support function calls (including recursive calls) in cutpoint proofs, but unlike in Axe, loops are not represented as function calls.

Of course, Axe is also used to perform proofs connecting equivalent loops in different programs; these are somewhat similar to the "double cutpoint" proofs described in the next section.

3.4 Translation Validation

A prominent application of equivalence checking for software is "translation validation" [53], which involves checking each step of a compilation or synthesis process as it is performed. For example, a verifying compiler could use translation validation to prove the equivalence of the input and output of each compilation pass, as was done by Necula for the GNU C Compiler [49]. Conceptually, translation validation is an easier problem than that solved by Axe. Axe applies to programs written totally independently, and much of the challenge in Axe's verification process is to discover (using test cases), how two programs correspond. By contrast, translation validation typically involves checking the operation of a known translator, which should be able to produce information to assist the validation process. For example, an optimizing compiler "knows" what optimizations it is performing and should be able to indicate to the validator how the input and output of each compiler phase match up. Also, Axe often must deal with several transformations at once, whereas translation validation can be performed after each phase of a multi-phase compilation process. Verifying each change to the program separately should make the individual proofs easier, since the difficulty of equivalence checking scales with the amount of difference between the programs being compared.

Among the most prominent translation validation tools is TVOC [9]. TVOC divides the program transformations it handles into two types: structure-preserving transformations (such as dead code elimination and constant folding) and structure-modifying transformations (which change the loop structures of program). This separation of concerns mirrors how Axe handles loops: loop transformations are handled first, and then, once the programs are synchronized, equivalence checking can be done using a mapping between corresponding loops. When checking a structure-preserving transformation, TVOC uses a data mapping between the variables in the two transition systems representing the programs being compared. Such mappings are analogous to Axe's "connection relations" linking the variables of equivalent loops. TVOC also uses a control mapping between corresponding source locations. It then performs what can be thought of as a "double cutpoint" proof (using a proof rule called Val) that requires checking that the data mapping is preserved as the two programs execute along simple paths between corresponding cutpoints. If all simple paths between cutpoints preserve the data mapping and the control mapping, then, by induction, the programs are equivalent (in the sense indicated by the mappings). Such a proof can be considered a two-program analogue of the standard cutpoint proofs described above.

TVOC also handles structure-modifying transformations (loop transformations), and it does so using a different proof rule, called Permute. The Permute rule can deal with transformations that reorder the particular executions of statements in the loop body. For example, the loop indexing scheme in the original program may be such that the loop body is executed first with i=1 and j=2 and later with i=2 and j=1. After re-indexing the loops, those executions may occur in the opposite order. To verify the transformation, one must check that such reordering does not change the observable behavior of the program. That is, any operations whose order is reversed by the transformation must be shown to actually commute. The Permute rule requires a mapping between particular iterations of the old and new loops, and this is represented in an "optimization spec" produced by the compiler (giving the details of the transformations it performed). If no mapping is provided, TVOC can in some cases use heuristics to discover one. This method of handling loops is quite different from what Axe does, since it handles a general class of loop transformations and uses a special rule of inference (Permute). By contrast, the Axe system has a built-in set of loop transformations that it performs[7]. For each transformation, Axe builds a new loop function and then proves the new and old functions equivalent using a standard (but completely automatic) ACL2 induction proof. No new rules of inference are used.

[7] Future work includes extending this set, or perhaps adding support for something like TVOC's Permute rule.

3.5 Dynamic Program Analysis

Axe's induction proofs about loop functions require loop invariants, and Axe detects these (and other properties, such as connections between equivalent loops) by analyzing execution traces. Because it detects properties by running test cases, rather than by examining the structure of the programs, Axe's invariant generation is said to be "dynamic." Of course, other dynamic invariant generation tools exist, most prominently the Daikon system [25], which has been applied mainly to Java programs but can also analyze data from other sources.

Daikon is based on a guess-and-check paradigm: it instantiates all invariants of the indicated shapes with all possible variables, drops any invariant that fails to hold on any test, and reports as candidate invariants any facts that are deemed to be statistically significant[8]. Axe does not explicitly instantiate all invariants of a given shape. However, it does consider all pairs of variables when attempting to express each variable in terms of any others.

In Daikon, the values assumed by a variable seem to be treated as a flat set of "samples." By contrast, Axe pays attention to how the samples are grouped in traces, as well as the order in which values appear in traces. For example, Axe checks whether a loop variable takes on the same sequence of values on every trace. (If so, it generates aggressive numeric bounds.) Axe also checks whether the values within a trace are monotonically increasing or decreasing. Another difference between Axe's analysis and Daikon's is that Axe is designed to find loop invariants, while Daikon is more targeted to finding preconditions and postconditions of subroutines[9]. Also, a major feature of Axe is its ability to find connection relationships between two

[8] Optimizations, such as not considering invariants that are implied by others, can make this process more tractable, but they are tricky; if you drop invariant x because y and z imply it, but y later turns out to be violated, you must start considering x again.

[9] One could force Daikon to find loop invariants by adding dummy subroutine calls in appropriate places.

programs. Daikon could probably be made to do this as well, perhaps by combining the two programs into one, or by combining the programs' data traces before analyzing them. Finally, much of Axe's invariant generation is based on the concept of finding "explanations" (ways to express quantities, such as the values of variables, or their upper or lower bounds, in terms of other variables), and Axe takes pains to avoid circularity in such explanations (because they will later be used as assumptions during rewriting). Axe could perhaps be made to use the output of a tool such as Daikon, but such issues would have to be dealt with.

The success of any dynamic invariant detection approach will depend on the quality of the test cases used. Luckily, for cryptographic programs, complex constraints on the inputs are rarely needed, but Axe does support the generation of random values that depend on the values of other inputs (e.g., generating a random input and then an output buffer of the same length). Axe also supports biasing of its random test cases in favor of classes of test cases that are more likely to reveal interesting behavior. (Section 8.3.1 describes Axe's random test generator.)

3.6 Static Program Analysis

Static techniques also exist to generate invariants and to prove properties of programs. Many tools focus on proving relatively shallow properties of large programs [60] [11], such as the absence of run time errors (buffer overflows, division by 0, dereferencing null points, violations of locking disciplines, etc.). Such tools can often handle very large programs (millions of lines), but, to be scalable, their analyses are necessarily imprecise, Thus, "false positives," error reports where no true errors exit, are a problem. In contrast, Axe proves much deeper properties (full functional correctness, stated as bit-for-bit equivalence of the outputs) but applies to much smaller programs (hundreds of lines). Since Axe performs equivalence checking, it is limited to verifying programs for which at least one additional implementation (or a formal specification) of the same algorithm can be obtained.

Many static program analyses follow the general technique of abstract interpretation [18] [19] [34]. In such a scheme, one picks a particular domain, such as intervals or polyhedra (representing conjunctions of inequalities over the program variables in n-dimensional space), and the abstract interpretation process provides a sound over-approximation of the properties of the program expressible in that domain[10]. A suitable use of abstract interpretation would likely generate many of the linear inequalities about loop indices that Axe uses during inductive verification. (One complication might be that Axe deals with Java bytecode, in which mathematical operations are performed in fixed-width fields, making "wrap around" a possibility.) The main reason that Axe does not use static analysis is that some of the properties that it needs would be very hard for any kind of static analysis to detect. For example, when comparing two loops, Axe often detects that pairs of variables always contain the same data values. These equalities may be true for very non-obvious reasons, e.g., because both loops compute one round of a cipher operation. The process of proving such facts may require significant work (e.g., unrolling inner loops and performing equivalence checking on the resulting DAGs). It seems unlikely that a sound static analysis tool would be able to prove such properties (unless it implemented many of the techniques used by Axe). On the other hand, many such properties are completely obvious when one simply examines execution traces: the two variables will always have the same random-looking value on every test case. Also, Axe already has test cases available for dynamic analysis, because it uses them to identify probably-equal nodes of miters.

[10] Such a scheme loses precision in two ways. First, the choice of the domain may cause information loss when constraints are combined for program paths that merge at "junction nodes" (e.g., when computing the convex hull of two polyhedra). Second, in order to make the process terminate, "widening" must sometime be performed; this can involve relaxing or discarding problematic constraints.

3.7 Verification of Cryptographic Code

In addition to the Cryptol work discussed above, a variety of approaches have been tried for verifying cryptographic code. However, as discussed in Section 1.1, the state of the art in practice is still testing, such as that performed under the auspices of NIST. Few real-world cryptographic implementations have been formally verified.

Others have verified block ciphers by proving the inversion property: that encryption followed by decryption with the same key produces the original plaintext. For example, Duan et al [22] used an interactive theorem prover to prove the inversion property of several ciphers expressed in higher-order logic. Their approach seemed to require significant manual interaction. (Also, it does not target pre-existing implementations like the Java programs verified by Axe.) More importantly, inversion proofs, while valuable, do not guarantee correctness. First, many insecure ciphers satisfy inversion. Consider a cipher for which encryption and decryption simply return their inputs unchanged (without using the key at all). Such a cipher satisfies inversion but is clearly not secure. Also, inversion proofs often fail to verify a block cipher's "key expansion" operation. Many block ciphers, including AES, take the key that is passed in and use it to generate a larger "expanded" key for use during the actual encryption operation. The same key expansion is performed when decrypting. Since the key expansion is identical for both encryption and decryption, an error in key expansion may exist in a cipher that satisfies inversion. (Perhaps a better approach would be to cross-check two cipher implementations by proving that the decryption operation of one cipher correctly inverts the encryption operation of the other, and vice versa.) The approach taken by Axe is to prove directly that an implementation agrees bit-for-bit with a trusted, formal specification for that algorithm, thus reducing verification to an equivalence checking problem.

The ECHO tool has been used to verify an implementation of AES [62] [61].

The ECHO approach involves using refactoring to undo optimizations in the code, thus synchronizing the program with its specification. User guidance seems necessary to identify the transformations to be performed (such an instances of data packing or patterns encoded in lookup tables), but some transformations can be detected automatically. In Axe, some user guidance (indicating loop transformations to be performed) is also necessary (but could perhaps be automated). However, when applied to unrollable algorithms, like AES, Axe requires no user annotations.

Post and Sinz applied the bounded model checker CBMC to prove functional equivalence of AES implementations [54]. The model checker was able to automatically verify the first three rounds of encryption, but a manual induction proof was needed to extend the proof to cover all rounds. Toma and Borrione verified a hardware implementation of the SHA-1 cryptographic hash function. The proof used the ACL2 interactive theorem prover and seemed to require significant manual effort [59].

3.8 ACL2

The Axe system is built on top of the ACL2 theorem proving system and borrows many ideas from it. Axe is less general, being targeted to equivalence checking of certain kinds of programs, but it is designed to provide much more automation for its proofs than ACL2 would provide.

The proofs done by Axe could also have been performed manually in ACL2, but with much more user interaction required. There are essentially two classes of proof in Axe: those for completely unrollable computations, and those that require induction. Axe's DAG representation is critical to its ability to represent unrolled cryptographic computations. Because the corresponding trees would be too large for ACL2 to deal with, ACL2 would not be able to verify cryptographic building blocks using the complete unrolling strategy. Thus, ACL2 proofs about those algorithms would likely be done by induction. This would be more labor intensive, because it would require formulating inductively-strong loop properties.

Axe also provides automation for induction proofs, by using test cases to propose likely invariants. In some cases, such as for the RC4 stream cipher, Axe can detect all properties needed to perform the verification automatically. In other cases, such as for the full proofs of MD5 and SHA-1, some user annotations are required[11]. Still, Axe detects many simple invariants of those programs. It would be tedious (and error-prone) to have to determine such properties by hand, as would be necessary for a manual ACL2 proof. Clearly, Axe cannot find all possible loop invariants that many be necessary in proofs about programs. The tradeoff is to sacrifice generality (compared to hand proofs done in ACL2) but receive greater automation for the programs that Axe can analyze.

Many parts of the Axe system are related to the corresponding components of ACL2. Most prominently, the logics of the two tools are the same (except for Axe's crucial addition of DAGs), as described in Chapter 4, and the Axe Rewriter was directly inspired by the ACL2 rewriter, as described in Chapter 5, which compares the two tools in detail.

Finally, Axe adds to ACL2 the important capability to call the STP prover on bit-vector goals. When reasoning about tricky cryptographic code (especially optimized code), this ability can be very helpful; STP can often quickly prove goals for which Axe lacks sufficient rewrite rules. Experience with Axe has shown that, while rewriting is very helpful in simplifying terms, it is undesirable to rely on rewrite rules to handle every last bit of reasoning required in a proof, especially when more automatic techniques are available.

[11] The generation of these could perhaps be automated, using information from the execution traces.

Chapter 4

Representing and Reasoning About Computations in Axe

There are many ways to represent computations, including source code, object code, hardware description languages, flow graphs, static single assignment form, etc. Axe's representation is based on the language of the ACL2 theorem prover, which is a side-effect-free dialect of Common Lisp. The ACL2 language is simple, mathematically precise, convenient to prove theorems about, and general enough that other representations (such as Java bytecode programs) can be translated into it. Axe adds to the language a more compact term representation in which structure is shared between common subterms; this structure sharing is crucial to Axe's ability to deal with the large terms that result from unrolling the loops in cryptographic code.

Axe's representation of computations allows equivalence checking of implementations written in different languages. Each implementation is first translated to the Axe intermediate representation, and all of the work of the equivalence check is performed on the Axe representations. This separation of concerns allows tricky semantic questions of a particular language to be dealt with up front, instead of interfering with the equivalence proof. Axe currently supports translation of Java bytecode programs to the Axe language in two ways, through symbolic execution (described in Section 9.3), and through decompilation (Section 9.4). Formal specifications are written directly in ACL2 and can thus be trivially converted to the Axe language[1]. Axe could be extended to apply to other programming languages, or even to hardware representations, simply by implementing a translator from the desired implementation language into the Axe language.

This chapter describes how computations and theorems proved about them are represented in ACL2 and how Axe extends this representation with DAGs. It also discusses how Axe's built-in operators are used to model a variety of cryptographic computations, and how formal specifications of cryptographic algorithms are written. Readers wishing to jump ahead to discussions of Axe's reasoning engines can skip this chapter and refer back to it as needed.

4.1 Values

The ACL2 logic [35] includes a subset of the values used by Common Lisp. These include numbers, strings, characters, and symbols (including the special symbols `t` and `nil`, which represent the boolean values "true" and "false"). Of these, Axe primarily uses booleans and numbers (especially the non-negative integers, which are used to represent bit-vectors). Sections 4.10.1 and 4.10.2 describe the boolean and bit-vector operations that are built in to the Axe system.

ACL2, like Common Lisp, also supports complex linked structures created by applying the function `cons`. The application of `cons` creates a "pair" whose individual values can be accessed using the functions `car` and `cdr`[2] The arguments of `cons` can themselves be calls to `cons` (but in ACL2 `cons` structures cannot be circular). Structures created by `cons` are also called "lists," and they include the "proper lists"

[1] One simply takes a call of the main specification function and makes it into a single-node DAG.

[2] These names are used for historical reasons.

for which taking successive `cdr`s eventually yields the value `nil`. Axe uses proper lists to represent tuples, arrays, and sequences, as described in sections 4.10.4, 4.10.6, and 4.10.5.

4.2 Terms

In ACL2, a term is a tree of variables, constants, and applications of operators (functions). Variables are symbols such as `x` and `plaintext`. Constants are preceded by a single quotation mark and include integer constants, such as `'17`, constants that are symbols, such as `'x` (different from the variable `x`), and constant sequences, such as `'(1 2 3 4)`, which is the sequence (or list, or array) containing the numbers 1 through 4[3].

Operators are function symbols (or closed lambda expressions, as described below). An operator is applied to its argument terms, with the entire application surrounded by parentheses. For example, `(cons '17 x)` represents the application of the function `cons` to the constant `'17` and the variable `x`. In ACL2, every operator has a fixed arity (e.g., `cons` always takes two arguments)[4].

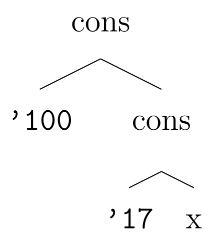
It is natural to view ACL2 terms as trees. For example, the term

```
(cons '100 (cons '17 x))
```

[3] The use of a single quotation mark preceding a constant is actually shorthand for a call of the special operator `quote`. Thus, `'4` and `quote 4` are the same.

[4] An operator which does not have a fixed arity can be implemented as a macro. For example, the operator `+`, which usually takes any number of arguments, macro-expands to a nest of calls to the function `binary-+`.

represents the tree:



Given an assignment of values to the variables in a term, one can compute the value of the term by simply evaluating the operators in a bottom-up fashion. For example, if `x` is assigned the value `'(1 2 3 4)`, then the value of `(cons '100 (cons '17 x))` is the constant `'(100 17 1 2 3 4)`. Two terms are considered equivalent if they have the same value for any assignment to their variables.

Functions in ACL2 never have side effects. Thus, ACL2 terms are referentially transparent; a subterm can be soundly replaced by any equivalent term without affecting the value of the overarching term; this property allows "rewriting" to simplify or transform terms while preserving their values (as described in Chapter 5).

A function symbol appearing as an operator in an ACL2 term can be one of the 32 ACL2 primitive functions, or any function built from them using ACL2's definitional principle (see Section 4.5), including a recursive function used to model a loop. Thus, ACL2 terms can represent computations on unbounded amounts of data.

4.3 Lambda Expressions and Beta-Reduction

The operators appearing in terms can also be closed lambda expressions. For example the expression:

```
(lambda (x y) (binary-+ x y))
```

represents the function that takes two arguments and applies the function `binary-+` to them. Every lambda expression in ACL2 is "closed," meaning that all the variables appearing in the body (`(binary-+ x y)` in the example) appear in the list of parameters (`(x y)` in the example).

Lambda expressions can be applied to arguments, just like regular function symbols. For example

```
((lambda (x y) (binary-+ x y)) '3 '4)
```

represents the application of the lambda expression to the arguments `'3` and `'4`. It's value is `7`.

Lambda expressions represent a very common operation in programming: binding a value to a name so that it can be referred to later in multiple places. (In ACL2, the macros `let` and `let*` expand into terms involving `lambda`.) Lambda expressions provide limited structure-sharing in ACL2 terms. For example, the term

```
((lambda (x) (foo x x)) (bar y))
```

is equivalent to

```
(foo (bar y) (bar y))
```

Notice that the call to `bar` appears twice in the latter but just once in the former. The process of expanding away a lambda expression is called beta-reduction. In general, beta-reduction can cause an exponential explosion in the size of the terms involved, due to the massive replication of identical subterms. If the call to `bar` in the example above were instead some huge term, beta-reduction would almost double the term size. Expanding a nest of lambda expressions can lead to repeated doubling, an exponential process.

Unfortunately, `lambda`s are expanded via beta-reduction early in ACL2's proof process (and ACL2 provides no convenient way to reintroduce them later). Nor is there a way to prevent ACL2's rewriter from processing repeated subterms, such as `(bar y)` above, each time they appear. Thus on some examples ACL2's prover does an exponential amount of work (as a function of the number of unique subterms being processed). Massive sharing of subterms is not restricted to pathological examples; it is common in terms that represent unrolled cryptographic computations. Perhaps this is not surprising, given that such algorithms perform extensive mixing of their inputs. The ability to efficiently deal with such massively-shared terms is a major advantage of Axe and is described in Section 4.8.

4.4 Theorems

In ACL2, a theorem is simply a term that is valid, that is, that evaluates to true[5] for all assignments of values to its free variables. (In ACL2, unlike many other logics, there is no distinction between terms and formulas; an ACL2 "formula" is just a term whose value is interpreted as a boolean.) An example theorem is:

```
(defthm car-cons-fact
  (equal (car (cons x y))
         x))
```

Here `defthm` means "define theorem," `car-cons-fact` is the name of the theorem, and the theorem body is the term

```
  (equal (car (cons x y))
         x))
```

which is true for all values of the variables `x` and `y`. The body of an ACL2 theorem can be considered to be surrounded by implicit universal quantifiers for all of its free variables.

[5] As noted above, ACL2 follows the Lisp convention that any value other than `nil` is considered to be "true."

The ACL2 logic is untyped and total, so every function is defined for all possible values. Thus, theorems often contain hypotheses specifying that the values involved are of the correct types. For example,

```
(defthm cons-car-cdr-theorem
  (implies (consp x)
           (equal (cons (car x) (cdr x))
                  x)))
```

requires that `x` be a cons pair (for any other value, the theorem is vacuously true because the antecedent of the `implies` is false). Of course, hypotheses are not limited to expressing such "type facts."

Theorems in ACL2 include the ACL2 axioms (which characterize the primitive functions), "definitional axioms" for defined functions (see Section 4.5), and theorems established by the ACL2 prover, often with user guidance. The ACL2 prover supports a variety of proof techniques, including simplification using conditional rewrite rules, linear arithmetic, "forward chaining," "meta rules," and proof by induction. It is quite complex, and we don't describe it fully here. Axe contains its own theorem proving component, the Axe Prover, which operates on terms in the Axe intermediate language and is described in Chapter 7.

Besides being logical objects, many ACL2 theorems can also be used as equivalence-preserving rewrite rules, as described in Chapter 5. The basic principle is that, because the theorem is true for all values of its variables, any particular instantiation of the theorem is also true.

4.5 Defining Functions

The ACL2 logic includes 32 primitive functions that are not defined in terms of simpler functions. The basic properties of these functions constitute the axioms of the ACL2 logic. For example, the axiom

```
(defaxiom commutativity-of-+
  (equal (+ x y)
         (+ y x)))
```

expresses the fact that addition is commutative. Reasoning about the primitive functions is done using these axioms, and perhaps by executing the functions on concrete arguments[6].

ACL2 also supports the definition of new functions, which can call the primitive functions or other previously defined functions. The definition of a new function extends the ACL2 logic according to a "definitional principle." In particular, defining a function causes a "definitional axiom" to be added to the logic, specifying that any call of the function is equal to the appropriately-instantiated function body. For example, one can define the function `double` as follows:

```
(defun double (x)
  (binary-* 2 x))
```

This definition causes a new axiom, referred to using the "rule name" `(:definition double)`, to be added to the ACL2 logic. That axiom asserts:

```
(equal (double x)
       (binary-* 2 x))
```

That is, for all values of `x`, the call of the function, `(double x)`, is equal to the body, `(binary-* 2 x)`. This axiom allows ACL2 to reason about `double`. In ACL2, adding a definitional axiom in this way produces a "conservative extension" of the logic, meaning that the new axiom doesn't cause any new properties to become true

[6] ACL2's built-in linear arithmetic procedure can also be used to reason about primitive functions that deal with arithmetic.

about pre-existing functions. In particular, defining a function cannot cause the logic to become inconsistent.

The function `double` is non-recursive (because its body does not include a call to `double` itself). ACL2 also supports the definition of recursive functions[7] Indeed, most interesting functions are recursive, including functions used to model loops, as described in Section 4.11.

Defining a recursive function also causes a definitional axiom to be added to the logic. For example, the function `len` (which computes the length of a list) is defined as follows:

```
(defun len (x)
  (if (consp x)
      (+ 1 (len (cdr x)))
      0))
```

(Here `if` is ACL2's if-then-else operator. Its arguments are the `if` condition or test, the "then-branch" and the "else-branch.") The resulting axiom, `(:definition len)`, says that, for all `x`, `(len x)` is equal to the body `(if (Jonas x) (+ 1 (len (cdr x))) 0)`.

Adding the definitional axiom for a recursive function presents a challenge to the principle of conservative extension, because some recursive equations are contradictory. For example, the definition:

```
(defun foo (x)
  (not (foo x)))
```

is not admissible in ACL2. If permitted, it would give rise to a definitional axiom equating `(foo x)` with `(not (foo x))`. This is a logical contradiction and so would

[7] It also allows the simultaneous definition of mutually recursive cliques of functions. Mutual recursion is rarely used in the computations about which Axe reasons, but the code of the Axe tool itself includes many mutually recursive cliques.

provide the means to prove any claim at all. Notice that the function `foo` is non-terminating; attempting to execute it on a concrete argument would cause an endless computation. It turns out that termination is a sufficient condition for a function's admission to produce a conservative extension of the logic. Thus, ACL2 requires that a termination proof be performed for each recursive function. (This involves formulating an ordinal measure over the function's arguments and proving that the measure decreases over every recursive call. The decreasing measure justifies performing induction proofs about the function, because it guarantees that such inductions are well founded.)

Termination of recursive functions is an interesting and deep topic, but it is not a major concern of Axe. In fact, Axe allows certain measures to be omitted and certain termination proofs to be skipped. Thus, the results proved by Axe can be considered "partial correctness" results; Axe guarantees that its results hold if all of the functions involved actually terminate. Termination can then be proven by other means (or non-termination can be easily observed if the program ever executes an infinite loop). All of the formal specifications of cryptographic algorithms have been proven to terminate, and proving an implementation equivalent to such a specification provides strong evidence that it terminates as well.

Because every ACL2 function has a precise mathematical meaning which can ultimately be traced back to the ACL2 primitives[8], ACL2 functions can always be executed on concrete arguments (unlike functions in other languages, which may be defined using quantifiers). Axe relies on this executability because it uses dynamic analysis to guide its proofs (e.g., to find pairs of equivalent nodes, or to find likely loop properties).

[8] Actually, this is not true for "constrained functions" or functions defined with `defchoose`, but these are beyond the scope of this discussion.

In addition to the 32 primitives, many functions are pre-defined in ACL2 and are available whenever ACL2 is started. Users of ACL2 typically begin their sessions by including additional "libraries" of relevant function definitions (usually including many proved theorems as well). Axe contains a large library of such functions, which are considered to be "built-in" to the Axe system. They deal with simple data types, including bit-vectors, arrays, sequences, and booleans (all described in Section 4.10). Axe's formal specifications of cryptographic algorithms (described in Section 4.12) are defined in terms of these built-ins. The built-in functions are also used to model the Java Virtual Machine and appear in the output of the JVM bytecode decompiler (Section 9.4). In addition, Axe may define new functions as it runs (e.g., to represent transformed versions of the loop functions being analyzed, or to represent loop conjunctions of invariants).

The ACL2 logic is "total" in the sense that every function is defined for all possible arguments, including arguments that seem not to be of the appropriate types. This avoids the need for "type correctness conditions" like those generated by the PVS theorem prover [32]. Axe's built-in functions usually begin by coercing their arguments to be of the correct type (for example, treating non-bit-vector arguments as 0). Thus, many theorems are true for all values, even those of the "wrong" type. Of course, theorems may also contain hypotheses asserting that the values involved have the correct types.

The preceding discussion of function definition applies to functions defined in "`:logic` mode." Proving an ACL2 theorem about a function requires that it be defined in `:logic` mode, and Axe makes the same restriction. However, ACL2 also allows functions to be defined in `:program` mode. One can execute such a function but not reason formally about it. Much of the code that constitutes the Axe tool itself consists of `:program` mode functions[9]. Both types of function can be compiled for efficient execution, and compilation is crucial to making the Axe tool fast enough to be practical.

4.6 Evaluating Terms

Axe includes the ability to evaluate terms on concrete arguments. That is, given a term and an association list that assigns values to all the variables in the term, Axe can compute the value of the term. This is possible even if the term includes functions that were defined after the Axe system itself was defined; definitions of non-built-in functions can be passed in to the evaluator. Evaluation uses two mutually-recursive functions, `apply` and `eval`, which operate as follows:

- To `eval` a constant, simply unquote it.

- To `eval` a variable, look up its value in the association list.

- To `eval` a term that is a function call (possibly a lambda application), `eval` the arguments and then `apply` the function to the evaluated arguments.

- To `apply` a primitive function to constant arguments, just call the function.

[9] The use of `:program` mode is required because Axe calls the external STP decision procedure using `sys-call`, which can only be done by `:program` mode functions. This issue would have to be addressed if the Axe tool itself were to be formally verified.

- To `apply` a non-primitive function to constant arguments, look up the function body, replace the formal parameters with the actual arguments, and `eval` the result.

Special handling is applied for `if` operators; the test is evaluated first and then only the relevant branch (then-branch or else branch) is evaluated. This special handling is critical to termination when evaluating calls of recursive functions[10].

The evaluator described so far is a standard "McCarthy-style" evaluator, but Axe's evaluator contains several extra features. First, for efficiency, the evaluator considers all of the Axe built-in functions to be "primitives," even though many of them have ACL2 definitions in terms of simpler functions. Thus, when asked to evaluate a call of `bvxor`, the evaluator simply calls `bvxor`, executing its compiled code, rather than interpreting it by opening up its body into more primitive functions (such as `logxor`[11]). This makes the evaluator much faster; the only functions which are actually "interpreted" are non-built-in functions, usually those that model loops.

Also, the evaluator for terms is actually mutually recursive with the evaluator for DAGs discussed in Section 4.9. This allows terms and DAGs to interoperate, as discussed in that section. Finally, the basic evaluator can be instrumented to produce "traces" capturing the values of function parameters on successive recursive calls. Analysis of the traces enables Axe to discover the loop properties that it will attempt to prove.

4.7 DAGs: Structure-shared Terms

Axe uses ACL2's representation of computations, with one key addition: a more compact representation of terms in which structure is shared between subterms that

[10] The evaluation procedure may still fail to terminate if asked to evaluate a recursive function on values for which the function does not terminate, but this rarely happens in Axe.

[11] which is not a primitive either.

are identical. Terms in ACL2 are simple trees, in which no structure is shared between identical branches, but Axe's representation is based on DAGs that allow sharing. The DAG representation is critical to Axe's ability to reason about unrolled cryptographic computations, because the terms representing such computations often have massive sharing. Since DAGs have a clear correspondence to terms, they share many of the nice features of ACL2's term language (clear mathematical meaning, referential transparency, etc.). The rest of this section illustrates the extensive sharing present in unrolled cryptographic computations and describes how Axe uses DAGs to represent them.

The key property of Axe's DAG representation is that a term may be shared by multiple "parent" terms. While this sharing could be accomplished by simply creating massively-shared linked structures in memory, it is helpful to have a way to print out and read in such terms. Doing so requires some abbreviated way to refer to shared terms (without repeating the entire term in full each time it appears). Axe's approach is to number each subterm. To represent a term as a DAG, one can simply assign consecutive, non-negative integers to each of its distinct subterms (constants, variables, and function calls) in such a way that identical subterms are assigned the same number, and with the numbers chosen such that the number of a function call term is always greater than the number of any of its arguments. For example, the subterms of the tree

```
(foo (bar x '3) (bar x '3))
```

could be assigned numbers as follows:

```
3:(foo 2:(bar 1:x 0:'3) 2:(bar 1:x 0:'3))
```

Note that both occurrences of the variable `x` are numbered 1, both occurrences of `'3` are numbered 0, and both occurrences of `(bar x '3)` are given the number 2.

The DAG representation of a tree is just an "association list"[12] mapping term numbers to their corresponding expressions, where arguments to function calls are replaced by the corresponding numbers. The DAG for the term above would then be

```
((3 foo 2 2)
 (2 bar 0 1)
 (1 . x)
 (0 quote 3))
```

Each entry in the association list is called a "node" of the DAG and pairs a node number with an expression. An "edge" in the DAG exists between a function call node and each of its argument nodes. The edge is considered to be directed (from the parent to the child), and, as mentioned above, the number of the child node must be smaller than the number of the parent. Thus, the graph is acyclic, because node numbers decrease along any edge. In the DAG above, the two edges corresponding to the call of `bar` go from node 2 to child nodes 0 and 1.

The association list representing a DAG is always sorted in decreasing order by node number; this allows the DAG to be quickly extended; new nodes can simply be `cons`ed onto the front of the list.

The DAG representation actually used by Axe contains one additional feature not mentioned in the above discussion: the arguments to a function can include quoted constants. So the DAG for the term `(foo (bar x '3) (bar x '3))` could simply be:

```
((2 foo 1 1)
 (1 bar 0 '3)
 (0 . x))
```

[12] For efficiency, Axe internally operates on DAGs as arrays, indexed by node number. This allows nodes to be looked up or changed in constant time. (ACL2, despite being side-effect-free, takes pains "under the hood" to ensure that destructive array operations are efficient, as long as "old" copies of arrays are never used.)

Here the constant `'3` appears “in line” as an argument to `bar`. Constants are very frequent arguments to function calls, since bit-vector widths and bit indices are usually constants. Representing these constants “in line” reduces the total number of nodes in the DAG and the number of nodes that must be looked up when processing the DAG.

The meaning of a DAG is clear because it is simply an abbreviated version of an ACL2 term. In particular, all operators are ACL2 functions and so have precise logical meanings. Unlike many other DAG-based representations (such as “AIGs”, DAGs for hardware circuits that contain only “and gates” and inverters), the operators in an Axe DAG can be any functions, including recursive functions.

When building a DAG, Axe evaluates “ground terms” whenever possible. That is, given a function call where all the arguments are constants (and where Axe has access to the definition of the function), Axe will execute the function on the constant arguments and insert the resulting constant into the DAG in place of the equivalent function call.

A DAG can be thought of as a huge, nested LET expression, with the LET binding for each node allowing its parents to refer to it by number. If a DAG node contains a function call, the function must a symbol (such as `bvxor`), never a lambda expression. Since the purpose of lambda expressions is to allow subterms to be named and referred to by name – which is already accomplished by node numbers – lambdas are not necessary and are “flattened” when DAGs are built. (Of course, the functions mentioned in a DAG can have definitions that mention `lambda` or `let`.)

The variables appearing in a DAG are considered to be its inputs, in the sense that, given values for a DAG’s variables, one can compute the values of all of its other nodes. This can be done simply, in a bottom-up fashion, starting with node 0. When a function call node is evaluated, its arguments will have already been evaluated, since they have smaller node numbers. The “value” of the DAG is taken to be the value of its “top” node (the node with the largest number).

Multiple terms can be represented, with maximal sharing, by using a single DAG; each term will then correspond to some node in the DAG. This is done, for example, by the Axe Prover (Chapter 7), which deals with disjunctions of potentially many terms. In the Prover, each disjunct is represented by a node number in a single overarching DAG, and subterms that appear in multiple disjuncts are thus represented only once.

4.8 Compactness of the DAG Representation

Because DAGs represent shared subterms only once, the DAG representation of a term may be smaller than the corresponding tree. Recall from above that the term `(foo (bar x '3) (bar x '3))` can be represented by the DAG:

```
((2 foo 1 1)
 (1 bar 0 '3)
 (0 . x))
```

Note that the `bar` node appears only once in the DAG but twice in the corresponding tree (and likewise for the variable `x` and the constant `'3`).

With larger DAGs, the effect of sharing can multiply. For example, the (32-bit) product of the `foo` term above with itself can be represented as:

```
(bvmult '32 (foo (bar x '3) (bar x '3)) (foo (bar x '3) (bar x '3)))
```

This is over twice a big as the original term. The corresponding DAG is:

```
((3 bvmult '32 2 2)
 (2 foo 1 1)
 (1 bar 0 '3)
 (0 . x))
```

which is only a single node larger than the original DAG.

As this example shows, adding a node to a DAG can correspond to a doubling of the size of the corresponding term. Adding n nodes can thus correspond to a multiplication of the term size by a factor of 2^n. Thus, the DAG representation can be exponentially more compact than the equivalent tree representation, if the term includes massive sharing.

In unrolled block cipher computations, such massive sharing does indeed exist. For example, table 4.1 shows the sizes of terms representing unrolled partial computations of the AES block cipher. Each term represents the execution of the given number of rounds of AES's inner encryption loop. Note that for each additional round, the size of the tree (total number of nodes) increases by a factor of about 13. By 10 rounds (the smallest number of rounds ever used in a complete encryption operation), the term contains more than 28 trillion nodes, which would be intractable to analyze. In contrast, the size of the corresponding DAG increases quite slowly. Each additional round adds 324 nodes to the DAG, and after 10 rounds the DAG contains a very manageable 3282 nodes.

Table 4.1: Representations of Partial AES Encryption

Iterations	Term Size	DAG Size
1	2633	366
2	35289	690
3	459817	1014
4	5978681	1338
5	77723913	1662
6	1010411929	1986
7	13135356137	2310
8	170759630841	2634
9	2219875201993	2958
10	28858377626969	3282

The Blowfish cipher provides an even more extreme example of DAG blowup. When represented as a tree, the unrolled version of Blowfish contains 220,639 DAG nodes (before bit-blasting), while the corresponding tree would require over 10^{5186} nodes. If we wish to use the unrolling approach to verify cryptographic algorithms (recall that this saves us from discovering and proving loop invariants), we need something like the Axe DAG representation.

Axe supports various operations on DAGs, including converting between terms and DAGs, merging two DAGs together into a single DAG (e.g., when building a miter to represent their equality), rewriting a DAG (Chapter 5), and performing equivalence checking on DAGs (Chapter 8). A key primitive operation when building DAGs is to determine whether a particular expression already exists in the DAG. If so, the existing node should be re-used. If not, the expression can be added without the risk of missing a chance for sharing. This operation corresponds to the "structural hashing" or "strashing" operation performed by most hardware equivalence checking tools. To support it, Axe keeps several auxiliary data structures when manipulating DAGs.

The DAG representation described in this section differs from ACL2's representation of terms, but it is sometimes necessary to use them together. In particular, Axe sometimes needs to define functions whose bodies are DAGs (because the corresponding trees would be too large). The next section describes the Axe Evaluator, which, in additional to being able to evaluate DAGs applied to constant arguments, also provides a way to embed DAGs inside ACL2 terms.

4.9 The Axe Evaluator

Axe contains an evaluator for DAGs which is similar to the term evaluator discussed in Section 4.6[13]. The Axe Evaluator is a function, `axe-evaluator`, that takes a DAG and an association list giving values to the input variables of the DAG. It

[13] In fact, the evaluators for terms and DAGs are mutually recursive.

returns the value of the top mode of the DAG. As for the term evaluator, the DAG evaluator accepts a list of definitions to enable it to evaluate functions that were not yet defined when the Axe system was defined (for example, functions that result from loop transformations performed by Axe). When a non-built-in function is encountered in a DAG being evaluated, its body is instantiated, yielding a term which is then evaluated using the term evaluator.

The Axe Evaluator gives special handling to each if-then-else node; it first evaluates the condition and then evaluates only the necessary branch. The nodes supporting the non-selected branch thus may not ever be evaluated (unless they have other parent nodes). The evaluator uses an array to track the nodes that have already been evaluated, so that it never evaluates a node more than once[14].

4.9.1 Embedding DAGs in terms

The Axe system relies on some important features of the underlying ACL2 system. For example, rather than re-implementing support for proof by induction, Axe simply calls ACL2 to perform such proofs. This means that all of the functions involved must be defined in the ACL2 logic. However, the function bodies that Axe needs to reason about may be too large to be represented efficiently as trees (e.g., if loops have been unrolled). Thus, it would be desirable to be able to define an ACL2 function whose body is a DAG. ACL2 of course does not allow this directly, but it can be done by using the Axe Evaluator to embed DAGs into ACL2 terms. For example, consider the term:

```
(bvmult '32 (bvplus '32 x y) (bvplus '32 x y))
```

[14] For technical reasons, the evaluator takes an argument that tracks the depth of arrays involved in the evaluation. (A DAG may contains a function whose body is another DAG, and so on.) The array depth parameter is logically irrelevant but allows the array operations to be fast.

in which the variables x and y appear freely. An analogous ACL2 function can be defined simply:

```
(defun f (x y)
  (bvmult '32 (bvplus '32 x y) (bvplus '32 x y)))
```

but note that the bvplus term appears twice. For terms with massive sharing, defining a function in this simple way would be inefficient. An equivalent function can be defined by using a call to the Axe Evaluator on an "embedded" DAG, as follows:

```
(defun f2 (x y)
  (axe-evaluator
   '((3 bvmult '32 2 2)
     (2 bvplus '32 0 1)
     (1 . y)
     (0 . x))
   (acons 'x x (acons 'y y (empty-alist)))
   nil ;no function definitions needed
   0   ;logically irrelevant array-depth parameter
   ))
```

Note that the DAG is passed in as a quoted constant. This constant may represent a massively shared term, but it will be relatively small because it is a DAG; unlike f, f2 only contains one mention of bvplus. As usual, axe-evaluator is passed an association list mapping the variables in the DAG to values, but in this case the association list is not a constant. The body of f2 clearly needs to depend on x and y, and this is accomplished by having the association list bind the symbols 'x and 'y to the actual values of x and y passed in to f2. Thus, the body of f2 contains free instances of x and y, as required.

A DAG may contain a node that is call to the function `axe-evaluator` (or, more likely, to some loop function which calls `axe-evaluator`). The inner call of `axe-evaluator` would of course have an argument that is a DAG. This second, "nested" DAG may ultimately call `axe-evaluator` on another nested DAG, and so on. Such nesting is used to represent computations with nested loops. To support such a situation, the Axe Evaluator for terms is able to evaluate a call of `axe-evaluator` – by simply calling `axe-evaluator`. This is the sense in which the evaluator for terms depends on the evaluator for DAGs, making them mutually recursive.

4.10 Built-in Data Types and Functions

This section discusses Axe's rich set of built-in operators. The operators are all defined in terms of simpler functions (ultimately, in terms of the primitives of the ACL2 logic). Of course, the definitions may involve recursion (e.g., for Axe's `bvand` operator, which is defined in terms of the recursive `logand` function). Any uncertainty about the meanings of the operators can be answered by referring to their definitions or by evaluating the operations on concrete arguments.

Much of the work of implementing Axe was in proving a large number of properties of its built-in operators (to be used as rewrite rules during the proofs). The library of rewrite rules may be of use to others who need to reason about similar concepts, even if they do not with to use the Axe tool.

4.10.1 Booleans

In ACL2 and Axe, the symbols `t` and `nil` have special meanings; they represent the boolean values "true" and "false," respectively. Axe provides built-in operators for dealing with boolean values, including `not`, `booland`, and `boolor`. These operators always return either `t` or `nil`, but they interpret their arguments according to the Lisp convention that anything other than `nil` is considered to be “true.” The Axe functions `booland` and `boolor` are true functions, unlike ACL2’s `and` and `or` operators, which are macros that expand to expressions using `if`[15].

Axe uses the type recognizer `booleanp`, which tests whether its argument is either `t` or `nil`. All of the comparison operators, such as `equal` and `bvlt`, satisfy `booleanp`.

4.10.2 Bit-Vectors

Among the most important of Axe’s built-in operators are those that manipulate bit-vectors. A bit-vector is simply a sequence of bits (`0`’s and `1`’s). Programming languages such as Java and C use bit-vectors to represent integer values within a given range, and the data values manipulated by cryptographic algorithms are also usually represented as bit-vectors.

Axe represents bit-vectors as non-negative integers. More precisely, bit-vectors of width n are represented by integers in the range $[0, 2^n - 1]$. ACL2 (along with Common Lisp) supports integers of unbounded size (“bignums”), so there is no limit on the bit-vector sizes supported by Axe. This representation is used by other ACL2 libraries for reasoning about bit-vectors, including AMD’s RTL library [1], the Integer Hardware Specification (IHS) library [16][16], and the “Super IHS” library from Rockwell Collins [2].

Axe’s bit-vector operators are also largely compatible with the corresponding operators used by the STP decision procedure, which are in turn designed to correspond to operations performed by real-world programming languages. Machine “integers”

[15] In the case of disjunction, Axe’s representation has an advantage. Consider that the ACL2 expression `(or x y)` macro-expands to `(if x x y)`, which mentions `x` twice. This means that the subterm `x` will be processed twice any time the `or` term is processed. Axe’s `boolor` operator avoids this problem.

[16] Axe’s implementation of bit-vectors shares some code with this library.

are represented by Axe as bit-vectors of a given size (e.g., 32 bits), not as mathematical integers, and this representation is essential to the faithful modeling of real-world systems. Unfortunately, bit-precise modeling does impose a burden on reasoning: certain properties only hold in the absence of overflow. The rewrite rules for bit-vectors take overflow into account (as they must, if they are to preserve equivalence), and, when rewrite rules are insufficient, the STP decision procedure can be very helpful in proving claims involving bit-vectors.

An advantage of representing bit-vectors as integers (as opposed to, say, lists of bits) is that it allows for fast execution. (For example, Axe's `bvxor` operator is ultimately defined in terms of Common Lisp's `logxor` function.) A disadvantage is that it is not possible to interrogate a bit-vector constant to determine its width (e.g., the constant `0` can be considered to be a bit-vector of any width). This deficiency is mostly alleviated by the use of explicit width parameters in the bit-vector operators themselves.

Axe supports a variety of operations on bit-vectors, including slicing, concatenating, rotating, logical operations (e.g., XOR), and arithmetic operations (such as addition). It also supports comparisons in which bit-vectors are interpreted as either signed or unsigned numbers. To reason about bit-vectors, Axe contains an extensive library of hundreds of rewrite rules. Axe's bit-vector operators are described in detail in Appendix A.1.

4.10.3 Integer Operations

While most mathematical operations in cryptographic code apply to bit-vectors (and thus “roll over” if the result exceeds the bit width), unlimited precision operations are occasionally involved in the proofs. For example, adding an element to a sequence increases its length by 1. Since that operation does not roll over, the rewrite rule capturing that fact is expressed using the true mathematical `+` operation, which operates on Common Lisp integers of unbounded size. Axe contains rules to convert such operations to the corresponding bit-vector operations when it can be proved that no rollover will occur.

4.10.4 Tuples

Axe supports three data types that are represented as Lisp lists: tuples, sequences, and bit-vector arrays.

In Axe, a tuple is a proper (`nil`-terminated) list of values. The length of a tuple is fixed, and its components need not all be of the same type. For example, a loop function may compute several results (say, two bit-vectors and an array of bit-vectors) and return them all as a tuple of three elements[17]. Tuples are created with `cons`, and components of tuples are extracted with `nth`. No complex reasoning is necessary for tuples.

4.10.5 Sequences

Axe's sequences are also represented as lists. They are unlike tuples in that they are usually homogeneous, and their lengths are not fixed. They are accessed using standard list operations, such as `car`, `cdr`, and `firstn`. Sequence operations can be used to express important loop invariants about cryptographic programs. For example, one can say that the "first n" elements of two sequences are equal. Such an assertion might be written in another language using a quantifier, but neither ACL2 not Axe supports quantifiers. (Of course, this avoids the need for quantifier elimination in the reasoning engines.) The `prefixp` predicate, used to test whether

[17] Axe avoids using ACL2's built-in support for multiple values (`mv` and `mv-let`) because those functions carry syntactic restrictions on their use which complicate the transformations Axe performs.

one sequence is a prefix of another, appears in the key annotations used to verify the SHA-1 and MD5 hash functions (see Section 2.5).

4.10.6 Arrays of Bit-Vectors

Axe provides special support for sequences that represent one-dimensional arrays of bit-vectors. These correspond to the arrays used in the Java programming language[18]

The Axe array operators are described in Appendix A.2. A few important differences exist between them and the corresponding operators of the STP tool. First, Axe's arrays (like any object in Axe or ACL2) can be compared with `equal`. That is, Axe's theory of arrays is "extensional." STP's arrays cannot be directly equated, and this must be taken into account when Axe calls STP (see Section 7.5). Also, the length of an array in STP must be a power of 2. Axe makes no such restriction (indeed, the cryptographic algorithms with which it deals use arrays of a variety of sizes). When array sizes are not powers of 2, there is a possibility of out-of-bounds array accesses; Axe's array operators take this into account, as do the rewrite rules for arrays. Finally, Axe supports constant arrays (represented as constant lists). Since STP doesn't provide syntax for constant arrays, care is taken to translate a constant array into an equivalent set of constraints when calling STP.

Arrays of more than one dimension are more complicated; for example, each row of a two-dimensional Java array is a separate heap object (and the rows of an array can alias each other!). Such arrays are currently only partially supported by Axe; code that uses them can be symbolically executed, but it cannot be processed by the decompiler (unless the decompiler performs complete unrolling).

[18] Java arrays live in the heap, but arrays are objects, so all of the values in an array exist at the same heap address. This greatly simplifies the alias analysis necessary to extract the computation performed by a Java bytecode program.

4.10.7 Pseudo-Higher-Order Functions

The ACL2 logic is first order, so functions cannot be passed as arguments to other functions. Yet a common operation in cryptographic code is to apply a function to every element of a sequence (for example, when “packing” data values, as described in the next section). Axe provides a facility for conveniently expressing such operations (in the formal specifications of cryptographic algorithms) and for reasoning about such operations when performing equivalence proofs.

Other languages provide higher-order functions, such as `map`, which takes a function and a list and applies the function to every element of the list. Such a `map` function is considered a “higher-order” function because one of its arguments is itself a function.

Axe provides a facility that uses ACL2 macros to achieve much of the functionality of higher order functions within ACL2’s first order logic paradigm. This facility is called `defmap`. For example, if one has the function `inc`, which simply increments its argument:

```
(defun inc (x) (+ 1 x))
```

and desires a function that applies `inc` to every element of a list, one can submit the following form:

```
(defmap map-inc (x) (inc x))
```

Here, `defmap` is a macro which causes several things to happen. First, a “map” function is defined which maps the supplied function over every element of a list. In this case, that new function is `map-inc`:

```
(defun map-inc (x)
  (if (atom x)
      nil
```

```
       (cons (inc (car x)) (map-inc (cdr x)))))
```

which walks down its input list, incrementing each argument. For example, `(map-inc '(1 2 3))` evaluates to `(2 3 4)`. But `defmap` does much more: it generates and proves 15 theorems about the new function. For example, it proves that the length of the list returned by `map-inc` is the same as the length of its input:

```
(defthm len-of-map-inc
  (equal (len (map-inc x))
         (len x)))
```

Also, it proves that the nth element of `map-inc` is simply `inc` applied to the nth argument of the input (or `nil` is n is too large):

```
(defthm nth-of-map-inc
  (implies (natp n)
           (equal (nth n (map-inc x))
                  (if (< n (len x))
                      (inc (nth n x))
                      nil))))
```

Thirteen other similar theorems are automatically proved by `defmap`[19].

The `defmap` utility supports more complicated schemes in which the mapped function takes more than one argument. There is also a related utility, `defforall`, that deals with checking whether all elements of a list satisfy a given predicate. These utilities are similar to but independent of the "map" utility in Jared Davis's "osets" library [20]. The Axe utilities are fairly general purpose and are standalone; they may be of use to ACL2 users who are not interested in using Axe.

[19] The proofs are done by "functional instantiation" of the corresponding proofs about a generic map function.

4.10.8 Data Packing

Cryptographic algorithms often perform "packing," in which small pieces of data are combined into larger values. For example, the RC6 block cipher takes a key that is a sequence of 8-bit bytes but packs the bytes into 32-bit words before passing the key data to the core encryption algorithm[20]. Special packing functions can be used to concisely represent this operation.

For an algorithm (like RC6) that operates on a fixed amount of data, the data packing can be expressed without any loops or recursion (albeit more verbosely). However, some algorithms perform packing on inputs that are not of fixed size. For example, the MD5 hash function operates on a sequence of bits of essentially unlimited length and packs the bits into 32-bit words (after performing a complicated padding operation). Since the amount of data packed is not fixed, some sort of loop or recursive function (or other infinitary method) is needed to express the packing operation. Axe provides several operators for this purpose.

The simplest packing operator is `(packbv bvcount bvwidth bvs)`, which takes a list, `bvs` of length `bvcount`. Each element of `bvs` should be a bit-vector of width `bvwidth`. The result is a bit-vector of width `bvcount` $*$ `bvwidth` (with the more significant bits of the result coming from the elements closer to the beginning of `bvs`). For example, `(packbv 3 4 '(#b1111 #b0000 #b1010))` returns `#b111100001010`. The `packbv` operation can be considered to be "big endian;" if "little endian" behavior is desired, one can reverse `bvs` before calling `packbv`. (This is done in the packing for the MD5 hash function.)

The inverse operation of packbv is `(unpackbv bvcount bvsize bv)`, which divides its bit-vector argument, `bv`, into `bvcount` pieces, each of width `bvsize`. For example, `(unpackbv 3 4 #b111100001010)` returns `(#b1111 #b0000 #b1010)`.

[20] RC6 also supports word sizes other than 32 bits, but 32 is the most common.

The defmap utility described in the previous section is used to generate the built-in functions map-packbv and map-unpackbv which apply packing and unpacking to each element of their input lists.

Data packing also involves grouping data into chunks. For example, to pack a long sequence of bits into bytes, one could group the bits into groups of 8 and then call map-packbv on the list of groups. To support this, Axe provides the function group. For example, (group 4 '(0 0 0 0 1 1 1 1)) is '((0 0 0 0) (1 1 1 1)). Any partial groups are discarded. The function ungroup performs the inverse operation.

Packing operations are used extensively in the specifications of the cryptographic hash functions. For example,the specification of MD5 contains the following function:

```
(defun divide-into-words (msg)
  (map-packbv 4 8 (map-reverse (group 4 (map-packbv 8 1 (group 8 msg))))))
```

Here msg is a list of bits. The bits are first grouped into groups of 8, and each group is packed into a byte. Then the bytes are grouped into groups of 4, each group is reversed (to achieve the correct endianness), and each group of 4 bytes is packed into a word. (At a later stage, the words are grouped into 16-word blocks.) This example shows an advantage of keeping the grouping and packing operations separate; one can interpose the call to map-reverse after grouping and before packing to achieve the correct endianness.

The functions dealing with data packing are built in to Axe, and Axe contains lemmas for reasoning about them (See Section 6.3).

4.11 Modeling Loops With Recursive Functions

In addition to its built-in functions, Axe can reason about iterative computations that involve loops. Since ACL2 is a functional language, it lacks looping constructs such as for, or while. Instead, repetitive computations are represented using recursion.

Axe uses the same approach. In particular, this section describes how tail-recursive functions of a particular sort naturally correspond to the loops in an imperative program.

A typical loop in an imperative program operates on a finite set of "loop variables"[21]. A loop will typically contain an exit test, which, when true, causes the loop to terminate, and a body, which updates some or all of the loop variables. (For the purposes of this discussion, any initialization of the loop variables is not considered part of the loop.)

For example, consider this pseudocode loop from the official specification of the RC6 block cipher[22].

```
for i = 1 to 2r + 3 do
    S[i] = S[i-1] + Q32
```

The loop parameters are all of the variables read or written in the loop, namely, `i`, `r`, and `S` (`Q32` is a constant). Even though the variable `S` represents an entire array, it is considered a single loop parameter. The exit test of the loop is `i > 2r + 3` because, when this test becomes true, the loop terminates. The loop body consists of the updates to the loop parameters, namely the explicit update to the array `S` and the implicit increment of the loop parameter `i`.

A loop such as the one above can be simply modeled as a tail-recursive function. A tail-recursive function is a function which never performs any work after making a recursive call. Instead, the value of the recursive call is simply returned as the value of the function. Such functions are desirable in Lisp programming because they can be compiled into loops in a straightforward way, saving the overhead of recursive

[21] Here we are excluding loops that access unbounded amounts of storage by traversing or creating linked structures in the heap. But we include loops that manipulate arrays; an entire array is considered a single loop variable.

[22] This is the first of the two loops used to calculate the cipher's key schedule. The loop uses a constant Q_w, where w is the word size. Since 32 is the most common word size, that constant is rendered here as `Q32`. `Q32` is actually the hexadecimal number `9E3779B9`.

function calls. Axe exploits this correspondence in the opposite direction, turning loops into recursive functions for ease of reasoning.

The parameters of the tail-recursive function representing a loop will simply be all of the variables dealt with by the loop. The function will first evaluate the loop exit test[23]; if it is true, the function will exit, returning some or all of the loop parameters (whichever ones are used by the code that follows the loop). If the exit test is false, the function will update the loop parameters and then call itself recursively on the updated parameters. Using this scheme, the loop above can be modeled as the recursive function `keyloop1`:

```
(defun keyloop1 (i r S)
  (declare (xargs :measure (nfix (+ 1 (- (+ (* 2 r) 3) i)))))
  (if (or (> i (+ (* 2 r) 3))
          (not (natp i))
          (not (natp r)))
      S
      (let ((S (bv-array-write 32
                               (+ (* 2 r) 4)
                               i
                               (word+ (bv-array-read 32 (+ (* 2 r) 4) (+ -1 i) S)
                                      *q32*)
                               S)))
        (keyloop1 (+ 1 i) r S))))
```

In `keyloop1`, the exit test is:

```
(or (> i (+ (* 2 r) 3))
    (not (natp i))
    (not (natp r)))
```

which indicates that the function should exit when `i` finally exceeds `2r+3` or if either `i` or `r` has an unexpected value (anything other than a natural number). Upon exit,

[23] To ensure termination of the function, the exit test may also include checks that cause the function to exit on unexpected inputs, e.g., inputs that are not natural numbers when natural numbers are expected.

the function simply returns S (the purpose of the loop is to populate the array S). On the recursive call, the function updates S. (Axe's array operations carry explicit size parameters; since S contains 32-bit elements and has length 2r+4, these sizes appear as arguments to bv-array-read and bv-array-write.) The update of S consists of reading the value at index i-1 from S, adding it to the constant *q32* (here the addition is modulo 2^{32} and is represented by the call to word+) and writing the result back into S at index i. The updated array and the incremented value or i are passed to the recursive call.

The function keyloop1 represents just one of four loops (not counting packing functions) in the formal specification for RC6. The other loops are modeled similarly, as are the loops in the other formal specifications of cryptographic algorithms. The JVM bytecode decompiler also generates loop functions.

Throughout this dissertation, whenever we say that Axe reasons about "loops," we actually mean that it reasons about the corresponding recursive functions. Furthermore, most of the proofs and transformations applied by Axe are restricted to "simple loop functions." Like keyloop1, any simple loop function is tail-recursive and has a single recursive call. It has an exit test that selects between two branches, the recursive call and a base case. The base case performs no real work; it simply returns one or more of the loop parameters (either a single parameter or a tuple of parameters). Most of the work is done when computing the new values of the loop parameters that are passed to the recursive call. (A function may have its recursive call wrapped in calls to let and still be considered a simple loop function.) The "body" of a simple loop function consists of the set of expressions that update the values of each of its parameters. Occasionally Axe will encounter a function not in simple form, either in a formal specification or as a result of applying a loop transformation (e.g., a function with more than one base case, due to constant factor unrolling). In such a case, Axe will first attempt to transform the function into an equivalent simple loop function, as described in Section 10.2.

4.12 Formal Specifications

Axe includes a library of formal specifications of cryptographic algorithms, including block ciphers, stream ciphers, and cryptographic hash functions. The official definitions of the algorithms are typically informal documents which describe the algorithms using English prose and pseudocode. For example, the official specification of the AES block cipher is given in Federal Information Processing Standard 197 (FIPS 197), a document published by the U.S. government's National Institute of Standards and Technology [5]. The formal specifications in Axe were written by formalizing these informal descriptions. They are all written in the ACL2 language, giving a precise mathematical description of each cipher. In writing the formal specifications, the emphasis was on clarity and faithfulness to the official informal descriptions. In particular, the formal specifications do not include any complicated optimizations not present in the official descriptions; thus, the equivalence checking process must handle optimizations present in implementations.

Loops in the formal specifications are modeled as recursive functions, using the method described in the previous section.

Even though efficiency was not a consideration in writing the formal specifications, they take the form of normal ACL2 functions and so are executable on concrete inputs. In fact, they have been validated by execution on test vectors supplied in the informal algorithm descriptions. Despite the lack of optimizations, the specifications still run fairly fast. For example, evaluating the specification for AES-128 on a concrete plaintext and key produces the ciphertext in 0.01 seconds.

All of the formal specifications are ACL2 functions, and none of them use Axe's

DAG representation of terms[24]. Thus the specifications may be of use to other ACL2 users who wish to reason about cryptographic code (or wish to model larger systems that contain such code), even if they don't wish to use Axe.

[24] Of course, reasoning about them using the technique of complete unrolling requires such a representation in order to be efficient.

Chapter 5

The Axe Rewriter

Axe contains a sophisticated rewriter capable of efficiently transforming large terms by repeatedly applying local "rewrite rules." The rewrite rules used are almost all equivalence-preserving[1]. That is, they allow a term to be replaced only by another term which always returns the same value. Because Axe terms are side-effect-free, rewriting can be performed without changing the meaning of any over-arching term.

A major goal of rewriting is to simplify the terms involved. Simplification is almost always desirable; it helps keep terms small and is sufficient to completely prove some claims (because they simplify down to "true"). A related goal is to normalize terms. Normalization is the process of transforming syntactically different but semantically equivalent terms so that they have the same syntactic representation (and so are clearly equal by inspection). Efficient rewriting schemes may not exist to completely normalize the terms with which Axe deals[2]. Nevertheless, normalization works fairly well in practice, using the techniques described in this chapter and the rewrite rules described in the next chapter.

[1] Non-equivalence-preserving rewrite rules are used in the Java decompiler to eliminate execution branches that throw exceptions.

[2] Consider that Axe's operators include the boolean operators, and boolean satisfiability can be reduced to normalization of boolean expressions (a formula is unsatisfiable iff it has the same normal form as the formula "false").

Rewriting can also be used to simply change the form of terms, for example, to bit-blast them, or to eliminate certain operators (such as the `leftrotate` operator, which cannot always be directly translated into the language of STP). Finally, rewriting can be used to strengthen a set of known facts, using information provided by some new fact; this occurs when a miter is split into cases (see Section 8.4).

The Axe Rewriter is similar to the rewriter of the ACL2 theorem prover [42] and borrows many important ideas from it. However, the Axe Rewriter has several advantages. The most significant is its efficient representation of terms as DAGs. It also performs memoization, uses "objectives" to guide rewriting, efficiently handles large XOR nests, provides more fine-grained control of rule ordering, and provides a facility for calling the Axe Prover to discharge hypotheses of conditional rewrite rules.

The Axe Rewriter is designed to be used as a standalone component, in contrast to the ACL2 rewriter, which is interwoven with the rest of ACL2's proof process[3]. The standalone nature of the Axe Rewriter allows it to be called by the Axe system in many places, whenever it is helpful to simplify terms.

An alternative to using a rewriter in a verification system is to simply encode all the desired transformations by hand; one would simply write a program that transforms terms (DAGs) in whatever way is desired. However, this is likely to be complicated and error prone. Adding a new transformation would require writing more code, and programming bugs could compromise the correctness of the system. Axe's approach is to factor the problem into two parts, a general-purpose rewriter and a set of rewrite rules (theorems) used to "program" the rewriter. This approach has several advantages. First, it allows a large number of different transformations to be added to the system; Axe contains hundreds of rewrite rules, and it would be tedious to write code for all of them. Second, Axe's approach makes it easy to

[3] ACL2 does include a tool, called the "expander," which allows the user to call the rewriter directly [3]. However, the implementation is somewhat ad hoc, and no soundness claims are provided. In contrast, The Axe Rewriter is designed to be trusted.

experiment with different sets of transformations; one simply passes different sets of rewrite rules to the Axe Rewriter. Similarly, one can easily change the order in which transformations are applied, using Axe's system of rule "priorities" (see Section 5.15). Finally, adding new transformations can be done in a sound way because the transformations can be proved as theorems in the ACL2 logic. This ensures that adding a new rule won't render the system unsound. When new rules are added, the amount of the code that must be trusted (the code of the Axe Rewriter itself) stays the same.

The Axe Rewriter lacks several features of the ACL2 rewriter, including "forward-chaining" rules, a procedure to determine the types of terms, and a built-in linear arithmetic procedure. None of these features were necessary for the Axe proofs discussed in this dissertation.

Each section of this chapter covers a feature of the Axe Rewriter. When appropriate, comparison is made to analogous features of ACL2. Since ACL2 is a complicated system and is not the focus of this dissertation, this chapter occasionally oversimplifies ACL2's behavior. For the complete details about ACL2, see its documentation [36].

5.1 Evaluating Function Calls

Perhaps the simplest operation performed by a rewriter is to evaluate a function call whose arguments are constants. For example, the term `(< '2 '3)` would rewrite to the constant term `'t` because 2 is less than 3. A constant is considered to be maximally simplified.

Functions in ACL2 are almost always executable (in contrast to other theorem-proving systems, especially those which make extensive use of quantifiers), and ACL2's rewriter evaluates function applications unless the user disables this behavior.

(Disabling evaluation is necessary only in rare cases, when the results would be prohibitively large, such as evaluating `(repeat 4294967295 0)`, which would generate a list containing over 4 billion zeros.)

The Axe Rewriter also evaluates functions applied to constant arguments, using the term evaluator described in Section 4.6 (and with special treatment for some functions, such as `repeat`, which is only evaluated on small arguments).

5.2 Unconditional Rewrite Rules

A theorem in the ACL2 logic can be seen as both a valid logical formula and a rewrite rule denoting a legal transformation that can be applied to a term while preserving its logical meaning. In general, a rewrite rule has conditions (called hypotheses) and a conclusion. This section deals with unconditional rules (those without hypotheses and for which the rule's conclusion is the entire formula); conditional rules are described in Section 5.4.

The simplest rewrite rules have conclusions that are equalities. Consider the rule:

```
(defthm car-cons-fact
  (equal (car (cons x y))
         x))
```

This formula is valid (for all values of `x` and `y`, the `car` of `(cons x y)` is always `x`), and `car-cons-fact` is thus a theorem in the ACL2 logic. It can also be used as a simplification rule; any term that fits the pattern (an application of the function `car` whose argument is an application of the function `cons`) can be rewritten using `car-cons-fact`. The rewriting process would replace the entire call to `car` with just the first argument of the `cons`. For example, applying `car-cons-fact` to the term:

```
(car (cons (foo xx '17) (bar z (baz '9))))
```

would yield:

```
(foo xx '17)
```

This new term is clearly simpler; it is smaller, and it doesn't mention the variable `z`, which was irrelevant in the original term[4].

Rewrite rules can be thought of as directed equalities and so are said to have left and right "sides" and to transform the "left side" into the "right side." For `car-cons-fact`, the left side is `(car (cons x y))`, and the right side is `x`.

The process of rewriting a term T using an unconditional rewrite rule has two steps:

1. Semi-unify the left side of the rule with T, yielding a substitution which, when applied to replace the variables in the left side of the rule, would yield T.

2. If Step 1 succeeded, instantiate the right side of the rule using the substitution, and replace T with the result.

Applying `car-cons-fact` to the term:

```
(car (cons (foo xx '17) (bar z (baz '9))))
```

gives a substitution which maps `x` to `(foo xx '17)` and `y` to `(bar z (baz '9))`.

The semi-unification done in Step 1 is also called "one-way unification;" variables in the rule can be instantiated, but variables the the term cannot.

Of course, if a variable appears more than once in the left side of a rule, every occurrence of the variable must unify with the same subterm. For example, the rule

```
(defthm equal-same
  (equal (equal x x)
         t))
```

[4] Rules that throw away irrelevant terms are among the most helpful rewrite rules, especially if the discarded terms are large.

only unifies with equality terms both of whose arguments are the same. (Without that restriction, `equal-same` could be used to rewrite any equality to true – which is clearly unsound!)

Rewriting with a rule that has been proved correct is guaranteed to preserve the meanings of terms. Because the proof of the rule covers all possible ways of instantiating its variables, the specific instantiation used for any particular rewrite is sound.

So far we have only discussed rules whose conclusions are equalities. Axe handles several other forms of conclusions. Any conclusion of the form `(not <term>)`[5] is treated as the equivalent formula `(equal <term> nil)`. For example,

```
(defthm <-same
  (not (< x x)))
```

is treated as if it were

```
(defthm <-same
  (equal (< x x)
         nil))
```

Also, a rule whose conclusion is a predicate (a function that is known to satisfy `booleanp`) is treated as an equality of that predicate with `t`. For example, since `consp` always returns a boolean,

```
(defthm consp-of-cons
  (consp (cons x y)))
```

is treated as if it were

```
(defthm consp-of-cons
```

[5] Throughout this dissertation we use angle brackets to denote placeholders in ACL2 terms.

```
  (equal (consp (cons x y))
         t))
```

Note that this treatment is not sound for conclusions that don't always return booleans. For example, the theorem

```
(defthm sum-not-nil
  (+ x y))
```

is valid, since `+` always returns a number, which is "true" in the sense of non-`nil`. But

```
(defthm sum-true
  (equal (+ x y)
         t))
```

is not valid.

In Axe and ACL2, the left side of a rewrite rule must be a function call. (A left side that is a constant would make little sense, because a constant is considered to be already simplified, and a rule whose left side is just a variable would probably not be very useful and would be very expensive, because the variable would unify with every possible term.) This requirement allows an optimization of the rewriting process: instead of trying to apply every rewrite rule to every term, Axe (and ACL2) checks the top function symbol of the term being rewritten and only tries those rules whose left sides are calls of that function. To support this, rules are grouped by function symbol before rewriting begins; this can be considered a simple form of term indexing.

Even with term indexing, most rewrite attempts will fail because the rule fails to unify, often because the term does not have the required "skeleton" of function symbols. For example, a rule whose left side is `(car (cons x y))` will likely be tried on many applications of `car` and will fail on all of them for which the argument to `car` is anything other than a call to `cons`. The Axe Rewriter takes pains to make such unification attempts fail fast, by deferring the actual building of the substitution until the term being rewritten is found to have the right skeleton.

5.3 Definitional Axioms

As described in Section 4.5, every non-primitive ACL2 function has a definitional axiom which defines its meaning in terms of other functions (primitives or previously defined functions). That axiom can be used as an unconditional rewrite rule, to turn a call of the function into the appropriately instantiated function body. For example, the function `bvle` (“bit-vector less than or equal”) is defined in terms of `bvlt` (“bit-vector less than”) as follows:

```
(defun bvle (size x y)
  (not (bvlt size y x)))
```

This definition corresponds to the definitional axiom:

```
(equal (bvle size x y)
       (not (bvlt size y x)))
```

which can be used to transform:

```
(bvle ’4 (bvplus ’4 w v) z)
```

into the equivalent term:

```
(not (bvlt ’4 z (bvplus ’4 w v)))
```

For a non-recursive function that is not built in to Axe, using the definitional axiom to open the function (by expanding its definition) is often a good idea. For example, non-recursive functions in the formal specifications for cryptographic algorithms are almost always opened. For recursive functions, care must be taken to avoid looping rewrites. Because opening a call to a recursive function produces a term that includes another call to the function, the function definition may be opened repeatedly forever. Section 10.4.1 discusses how Axe deals with this issue.

5.4 Conditional Rewrite Rules

Conditional rewrite rules are those that have hypotheses. The general form is:

```
(defthm <name>
  (implies <hyp>
           <conclusion>))
```

for a theorem with a single hypothesis, or

```
(defthm <name>
  (implies (and <hyp_1>
                ...
                <hyp_n>)
           <conclusion>))
```

for a theorem with multiple hypotheses.

Here is an example of a conditional rewrite rule:

```
(defthm cons-of-car-and-cdr
  (implies (consp x)
           (equal (cons (car x) (cdr x))
                  x)))
```

This rule says that if `x` satisfies `consp` (which recognizes values built by calling `cons`), then `cons`ing the `car` of `x` onto the `cdr` of `x` gives `x`. The hypothesis is necessary; if `x` does not satisfy `consp`, the equality must actually be false (because such an `x` cannot be equal to a call of `cons`).

From a logical point of view, a hypothesis weakens a theorem. However, the hypothesis may be necessary for the theorem to be valid. (Or it may simply make the theorem easier to prove.) Hypotheses impose a burden during the rewriting process: each hypothesis must be "relieved." That is, in order for the rewrite to apply in a particular situation, the rewriter must establish that the hypotheses of the rule are true in that situation. This is done by rewriting the hypotheses. More precisely, the process of applying a conditional rewrite rule to a concrete term has three steps (ignoring for now the handling of certain special hypotheses mentioning `syntaxp` and `bind-free`):

1. Unify the left side of the rule with the term, yielding a substitution which, when applied to the left side of the rule, yields the term being rewritten.

2. If step 1 succeeded, relieve the hypotheses in order, by instantiating them using the substitution, rewriting the results, and checking that each one rewrites to a non-`nil` constant. (This ignores the issue of "free variables," which are variables present in a hypothesis but not bound by the substitution. They are discussed in Section 5.6.)

3. If all the hypotheses were relieved in step 2, instantiate the right side of the rule using the substitution and replace the original term.

The hypotheses of conditional rules are relieved using recursive invocations of the rewriting process. These rewrites can also involve conditional rules, leading to further invocations of rewriting. The whole process can be very expensive unless care is taken (as with the rewrite rules used by Axe) to prevent expensive occurrences of this sort of "backchaining."[6]

The order of the hypotheses of a conditional rule can affect rewriting performance. Consider a rule for which a great deal of work is performed to rewrite its first few hypotheses, only to have the entire rule application fail because some later hypotheses fails to be relieved. In such a situation, all of the work done to relieve the early hypotheses is wasted (unless some results computed and memoized during that process happen to be useful later). The rules used by Axe have been designed to be fast. That is, their hypotheses are ordered so that the ones that are cheap to relieve, or that are likely to fail, appear early in the list.

5.5 `syntaxp`: Syntactic Restriction of Rewrite Applications

The rewriters of both ACL2 and Axe allow syntactic filters to be attached to rewrite rules. Such a filter examines the structure of the concrete terms involved in a potential application of a rule and prevents application in situations where rewriting would be undesirable. Forgoing the application of the rule can prevent unfavorable situations, such as rewriting loops, the replacement of something simple with something more complex, or the performance of potentially expensive operations. Syntactic checks deal with surface-level issues, such as how many operators appear in terms, or whether terms are quoted constants. These properties do not respect the substitution of equals for equals. For example, the terms `(bitand '17 x)` and `(bitand x '17)` are equal (by commutativity) but are syntactically different; the latter term will have its

[6] ACL2 allows rules to be annotated with "backchain limits," which limit the number of recursive invocations of rewriting. This hasn't yet proved necessary for Axe but could be implemented for the Axe Rewriter in the future.

arguments swapped to put the constant first (if Axe is using the rewrite rule that enforces that normal form).

In ACL2, a syntactic test is attached to a rule by including it as a special hypothesis, where the syntactic test is wrapped in a call to "`syntaxp`." A hypothesis wrapped in `syntaxp` is very different from a normal hypothesis. Whereas a normal hypothesis deals with the semantic meanings of the terms involved and is relieved through rewriting, a `syntaxp` hypothesis deals with the syntactic forms of the terms and is relieved by evaluation of the indicated syntactic check. In essence, the variables inside the call to `syntaxp` are bound to the tree structures of the actual terms they represent. Then the form inside the `syntaxp` is evaluated. If the result is `nil`, the hypothesis fails to be relieved. Otherwise, the `syntaxp` hypothesis is considered to be relieved and further hypotheses (if any) are dealt with as usual. Consider the following rewrite rule:

```
(defthm bitand-commute-constant
  (implies (syntaxp (quotep y))
           (equal (bitand x y)
                  (bitand y x))))
```

This rule reverses the order of the operands in a call to `bitand` whenever the second operand is a constant (the function `quotep` recognizes a quoted constant). For example, the rule would transform the term:

```
(bitand (bitxor x y) '1)
```

into the equivalent term

```
(bitand '1 (bitxor x y))
```

because `'1` is a constant. In this case, the expression inside the `syntaxp` hypothesis, namely `(quotep y)`, is evaluated with `y` bound to the syntactic term `'1` and returns true because '1 satisfies quotep. By contrast, the non-constant term (bitxor x y) does not satisfy quotep, so the rule would fail to further transform (bitand '1 (bitxor x y))[7].

This example is very simple; it simply checks with whether a term is a quoted constant. However, much more advanced syntactic tests are possible. Some rewrite rules in Axe's library use such tests to determine whether a term is a call of a function that is known to return a bit-vector. Others examine the "sizes" of operand terms to decide whether to commute them, as part of a scheme that moves smaller terms to the front of nests of function calls. It is common for the terms inside syntaxp to include calls to user-defined functions, and these functions can be recursive and so can traverse large terms.

A normal hypothesis can make a rule easier to prove but incurs the burden of being relieved before the rule can soundly be used during rewriting. The situation with syntaxp is different. Logically, the function syntaxp always returns t, so including it as a hypothesis of a rule gives no help in proving the rule. But no semantic burden is imposed when the rule is used; it is always sound to consider a syntaxp hypothesis to be relieved. The syntaxp test simply provides a heuristic filter to restrict application of the rule in certain cases.

All of the hypotheses of a rule, including syntaxp hypotheses, are relieved in the order in which they appear. Thus, a syntaxp hypothesis can refer to "free variables" that have been bound by earlier hypotheses (see Section 5.6). Usually, syntaxp hypotheses are fast to relieve, since they only require evaluation, rather than the recursive calls to the rewriter that are required for normal hypotheses. Thus, it is often advantageous to put syntaxp hypotheses early in a rule's hypothesis list.

The Axe Rewriter includes a facility very similar to ACL2's syntaxp feature. This

[7] The bitxor subterm will have already been rewritten into normal form before the bitand term is ever rewritten.

feature, called `axe-syntaxp`, is used analogously to `syntaxp`; it wraps syntactic tests in the hypotheses of rewrite rules. Like `syntaxp`, `axe-syntaxp` is logically equivalent to `t`; the soundness argument is thus the same as for `syntaxp`. The major difference is that terms in the Axe Rewriter are represented as DAGs. Thus, syntactic tests must operate not on ACL2 terms (trees), but on parts of DAGs. Instead of being bound to a term, each variable inside a call to `axe-syntaxp` is bound to either the number of a node in the DAG or a quoted constant. In addition, the special variable `dag-array` is bound to the DAG (represented as an array of nodes). Thus, Axe's syntactic tests may traverse through large portions of the DAG. The Axe Rewriter uses its internal evaluator (described in Section 4.6) to evaluate `axe-syntaxp` hypotheses. Any user-defined functions used in `axe-syntaxp` hypotheses should be built into the evaluator or have their definitions passed in to the rewriter.

Because of the different representations of terms, a rule using `syntaxp` cannot be used directly by the Axe Rewriter. Instead, a version of the rule must be defined that uses `axe-syntaxp` and operates on DAGs. It would be nice, but perhaps very difficult, to automatically translate syntactic tests that operate on terms into analogous tests for DAGs. (However, certain very common `syntaxp` hypotheses, such as those that just call `quotep`, can be used by Axe without any modification.)

The ACL2 rewriter supports "extended" `syntaxp` hypothesis, which have access to several internal data structures (collectively called the "metafunction context"). Axe has no analogous feature, but it does allow rules to access one important piece of information, the "rewrite objective," as described in Section 5.13.

5.6 Free Variables

A rewrite rule can have "free variables," which are any variables that occur in its hypotheses or right side but do not occur in its left side. (Throughout this discussion, when we say a variable "occurs" in a term, we do not count variables that are bound by constructs such as `let`, `let*`, or `lambda`.) Consider the following rule:

```
(defthm bvlt-transitive
  (implies (and (bvlt size x z)
                (bvlt size z y))
           (equal (bvlt size x y)
                  t)))
```

This rule says that `x` is less than `y` if there is some `z` between them. The variable `z` is a free variable in this rule, because it appears in the hypotheses but not in the left side.

Because a rewrite rule is true for all values of its variables, a free variable can be soundly instantiated with any concrete term. Of course, in order for the rule to apply, all hypotheses, including those that mention free variables, must be relieved. In practice, there are two ways in which the ACL2 rewriter "binds" free variables to concrete terms during the application of a rule: by searching among known assumptions, and by considering special "`bind-free`" hypotheses. The rest of this section describes the binding of free variables by searching among known assumptions, and the next section describes `bind-free` hypotheses.

As the ACL2 rewriter attempts to apply a rule, it builds a mapping that binds variables in the rule to concrete terms. Unifying the left side of the rule with the term to be rewritten binds all of the variables in the left side. Free variables are then bound as the hypotheses are relieved in order. When a hypothesis, H, is reached that mentions one or more free variables that don't yet have bindings, the rewriter attempts to bind them by performing a search through the set of known assumptions. To do the search, H is partially instantiated, using any bindings already present for its variables. Then the known assumptions are searched for a term which can be obtained by instantiating the remaining variables (the "free" variables) of H. If such a match is found, it is used to bind the free variables in H. H then counts as having been relieved (because when fully instantiated it is equal to a known assumption), and the rewriter moves on to consider the rest of the rule's hypotheses. If the rest of the hypotheses are relieved (which may involve binding additional free variables), then the rule applies. Otherwise, the rewriter backtracks to H and tries any additional matches that may exist among the known assumptions (corresponding to other instantiations of H's free variables). If another match is found, the rewriter uses it to bind the variables in H and attempts to relieve the rest of the hypotheses. If no more matches are found, H fails to be relieved. (Because the backtracking process just described can be expensive, ACL2 allows the user to specify that only the first match be used, rather than all matches.)[8]

The Axe Rewriter binds free variables in much the same way as the ACL2 rewriter, except that, in keeping with its representation, variables are bound to node numbers in the overarching DAG (or to quoted constants), rather than whole terms. In particular, the Axe Rewriter supports the backtracking search described above. (Indeed, Axe omits the option to only use the first match, which seems to be rarely useful.)

In ACL2 it is possible for the right side of a rule to contain a free variable that is not bound by any hypothesis in the rule. In such a situation, when the rewriter instantiates the right side of the rule, it simply leaves the variable unchanged. This is sound (since a theorem is true for any values of its variables) but can lead to the introduction of new variables into proofs in surprising ways. In Axe, such a situation causes an error.

The "known assumptions" in the previous discussion are passed in explicitly to the Axe Rewriter, in addition to the term being rewritten. Such assumptions are typically

[8] This discussion omits a few special cases in ACL2's handling of free variables. For the full description, see the ACL2 documentation topic FREE-VARIABLES.

not rewritten, but they are transformed slightly to more closely match the hypotheses that occur in rules, as described in Section 5.11. Also, to make free variable matching efficient, Axe performs term indexing of the assumptions using their topmost function symbols.

5.7 `bind-free`: Programmatic Binding of Free Variables

Because a rewrite rule is true for all values of its variables, free variables in the rule can be soundly instantiated in any way at all, as long as the hypotheses of the rule can be relieved. Instead of searching the known assumptions to bind the free variables, it is sometimes desirable to specify an explicit procedure to be used to bind them. In ACL2, this can be accomplished by using a feature called `bind-free`. Like `syntaxp`, `bind-free` appears in a hypothesis of a rule, takes a single argument, and is logically equivalent to `t`. Also, it has a pragmatic effect when the rule is applied and operates on the syntax of the terms involved in the rewrite, not their semantic values.

When a `bind-free` hypothesis is encountered, its argument is evaluated, with all variables bound to the syntactic terms that they represent. The result must be either `nil`, indicating failure to relieve the `bind-free` hypothesis, or a substitution which provides bindings for some of the free variables in the rule. In the latter case, the substitution is used to extend the existing substitution. This is sound because the free variables can be bound in any way at all[9]. As with `syntaxp`, `bind-free` can be used to call a complicated function that examines the terms involved in the rewrite. This gives `bind-free` hypotheses significant expressive power. Still, `bind-free` hypotheses are usually relatively cheap to relieve, because they do not involve recursive calls to

[9] If the variables bound by a `bind-free` hypothesis play a role in the correctness of the rule, they will be mentioned in later hypotheses, which will still have to be relieved.

the rewriter.

Axe supports a feature, `axe-bind-free`, that is very similar to ACL2's `bind-free`. The major difference is that forms inside `axe-bind-free` must operate on the DAG representation of terms, rather than the tree representation of ACL2.

The most common use of `axe-bind-free` is to bind a variable that represents the size of a bit-vector term. This is accomplished by the function `bind-term-size`. Consider the following rule:

```
(defthmd getbit-too-high-is-0-bind-free
  (implies (and (axe-bind-free (bind-term-size x 'xsize dag-array))
                (<= xsize n)
                (integerp n)
                (unsigned-byte-p xsize x))
           (equal (getbit n x)
                  0)))
```

This rule expresses the fact that extracting the bit at index `n` from a value whose bit width is `n` or less always returns 0. For example, the term:
`(getbit '10 (bvxor '8 x y))`
is equal to 0, because 10 exceeds 7, and 7 is the highest bit index in the 8-bit `bvxor` term. When the rewrite rule is applied to `(getbit '10 (bvxor '8 x y))`, the `bind-free` hypothesis binds the variable `xsize` to `'8`, the size of the term `(bvxor '8 x y)`. The other hypotheses are then relieved, because `(<= '8 '10)` is true, 10 is an integer, and `(unsigned-byte-p '8 (bvxor '8 x y))` rewrites to true.

5.8 Chains of Rewrites

The Axe Rewriter applies rewrite rules exhaustively, until nothing more can be done (at which point the term is said to be stable). This can lead to long chains of rewrites in which a term is rewritten, then the new term is rewritten to another term, and so on for many steps. Such behavior occurs during symbolic execution; a term representing a machine state is repeatedly rewritten, with each rewriting step corresponding to the execution of a single instruction (see Section 9.3). If many instructions are executed, this can lead to a very long chain of rewrites. For instance, symbolically executing the application of AES-128 to arbitrary inputs involves stepping through 9919 instructions. The Axe Rewriter is designed to support such long chains of rewrites. In particular, it avoids recursive function calls that accumulate on the call stack when rewrites are pending; such calls could cause crashes due to stack overflows. The Axe Rewriter avoids this problem by managing the stacks itself (e.g., as lists of nodes being rewritten).

The approach of rewriting until stability can lead to "rewriting loops" in which stability is never reached, because two terms (or more) are repeatedly rewritten to each other. For example the two rules:

```
(defthm foo-to-bar
  (equal (foo x)
         (bar x)))

(defthm bar-to-foo
  (equal (bar x)
         (foo x)))
```

will cause a rewriting loop when applied to any call of `foo` or `bar` (unless some other rule intervenes, transforming the call to `foo` or `bar` into some other term). Much more complicated examples of rewriting loops are possible. Care was taken to prevent rewriting loops when designing the rewrite rules used by Axe (the use of "objectives" as described in Section 5.13 can help), but some loops may still exist.

5.9 Use of Equalities

When the Axe Rewriter is given assumptions that are equalities (or can be easily converted into equalities, as described in Section 5.11), it uses them to perform replacement during rewriting. When rewriting a term that appears on the left side of an equality, Axe simply replaces it with the right side of the equality. It is up to the user to ensure that doing so does not cause looping behavior.

5.10 Handling of XORs

The XOR operation is very common in cryptographic code, because it can be undone by simply performing another XOR. XOR is associative and commutative, and so the result of XORing several values together can be represented in many equivalent but syntactically different ways. These different but equivalent "XOR nests" must be dealt with during equivalence checking, and for cryptographic code the number of operands in the nests can be quite large (dozens or hundreds). Nests of associative and commutative functions can be normalized by standard rewrite rules (which always operate locally, only dealing with small parts of the DAG at any one time). However, when the number of operands in the nests is large, this process can become inefficient. (Using local rewrites to "sort" a nest by swapping neighboring operands amounts to performing a bubble sort on the operands. Since bubble sorting has complexity $O(n^2)$, whereas sorting can be done in $O(n \log n)$ time, using local rewrites is too slow to be practical for large nests.)

To efficiently handle large XOR nests, Axe breaks with its approach of using a general purpose rewriter programmed by verified rewrite rules. Instead, it contains hand-written code to normalize XOR nests. This code makes use of the associativity and commutativity of XOR as well as two additional properties, namely, that XORing a value with itself is 0, expressed by the rule:

```
(defthm bvxor-same
  (equal (bvxor size x x)
         0))
```

and that XORing a value with 0 has no effect, expressed by the rule:

```
(defthm bvxor-of-0-arg2
  (equal (bvxor size 0 x)
         (loghead size x)))
```

(The `loghead` call in the conclusion has no effect unless `x` is wider than `size` bits, in which case `x` is trimmed down.) The output of Axe's XOR normalization procedure is a nest in which all XOR operations have been associated to the right (so that the left child of an XOR is never an XOR[10]), and in which the nodes have been sorted (by node number in the overarching DAG), duplicate node pairs have been removed, and all constants have been combined into a single constant. The single constant is dropped if it is 0, and is otherwise moved to the front of the nest. Nodes are sorted by decreasing node number, so that if a new (high-numbered) node is added to the nest, its proper place will be close to the "front" of the nest. If nodes were instead sorted by ascending node number, inserting a high-numbered node (which would belong at the deepest position in the nest) would cause the entire nest to be rebuilt.

As XOR normalization runs, it aggressively removes pairs of duplicate nodes, in order to keep the nest as small as possible at all times.

If in the future Axe is applied to examples with large nests of associative and commutative functions other than XOR, custom code could be added to Axe to normalize nests of arbitrary associative and commutative operators. (It would likely still be beneficial to handle XOR separately because of its cancellation property.)

[10] This discussion assumes that all of the XOR operations involved have the same bit-vector size.

5.11 Transforming Assumptions

The assumptions passed in to the Axe Rewriter are used in two ways: to find matches for free variables in hypotheses, and for equality substitution. In general, when rewriting a term with respect to a set of assumptions, the assumptions themselves are not rewritten. However, some transformations are performed to make them more suitable for these two purposes.

For example, consider the assumption `(unsigned-byte-p '32 x)`. This is not an equality, so it cannot be used directly for equality substitution. However, `unsigned-byte-p` is a predicate, so this assumption is logically equivalent to the equality:
`(equal (unsigned-byte-p '32 x) 't)`. This slight transformation allows the Axe Rewriter to replace the `unsigned-byte-p` fact with `'t` whenever it appears in a proof.

A similar situation exists for negations. Consider an assumption of:
`(not (bvlt '32 x y))`
This is not an equality, but it is equivalent to the equality: `(equal (bvlt '32 x y) 'nil)`
(Such a transformation is always valid, even when the negated term is not a predicate.) Again, the slight transformation allows the Axe Rewriter to use the assumption for equality substitution.

Assumptions are also used to bind free variables in rewrite rules. The preferred forms of assumptions for this use are different from the forms preferred for equality substitution. Consider the assumption `(equal (unsigned-byte-p '32 x) 't)`. This could be used to bind the free variable in, say, the hypothesis `(unsigned-byte-p '32 freevar)`, except that the form of the assumption is slightly wrong. One could write different variants of the rewrite rule, perhaps one with the hypothesis `(unsigned-byte-p '32 freevar)` and one with `(equal (unsigned-byte-p '32 `freevar) 't)`, but this would quickly become unwieldy, especially for rules with more than one hypothesis. A similar situation exists for negations. The fact `(equal (bvlt '32 x y) 'nil)` does not quite match the hypothesis `(not (bvlt '32 x y))`. To address these issues, Axe transforms assumptions to their "preferred" forms before free variable matching. (The preferred forms are essentially the opposite of what is preferred for equality substitution: `<x>` is preferred to `(equal <x> 't)`, and `(not <x>)` is preferred to `(equal <x> 'nil)`.) Authors of rewrite rules should state hypotheses using this preferred form[11]

5.12 Outside-In Rewriting

The ACL2 rewriter operates in an "inside-out" fashion. That is, when a function call is to be rewritten, the arguments are rewritten before the function call term itself. The argument terms are also rewritten inside-out, and so on, recursively, down through the entire tree being rewritten. The effect is that rewriting is done first to the leaves, and then to trees of increasingly larger size, with the root of the tree rewritten last.

Inside-out rewriting can be inefficient in certain cases, especially when it causes work to be done on terms that will later be discarded when some overarching term is rewritten. For example, consider a term of the form:

```
(car (cons x <big-term>))
```

where `<big-term>` is some term that is very expensive to rewrite. Rewriting this term inside-out will first involve rewriting `<big-term>`, yielding after much work the term[12]:

```
(car (cons x <rewritten-big-term>))
```

[11] An exception would be the use of `(equal x 'nil)` to mean that `x` is the empty list. Using `(not x)` would be less clear, because presumably `x` would not be a predicate in this case.

[12] These intermediate terms may not really be constructed; they simply serve as a representation of the intermediate state of the rewriter.

Then the application of `cons` will be rewritten. Assuming no rules apply to rewrite the `cons`, we will still have:

```
(car (cons x <rewritten-big-term>))
```

Finally, the application of `car` will be rewritten using `car-cons-fact` (from Section 4.4), giving just `x`. Since `<rewritten-big-term>` will be discarded by the application of `car-cons-fact`, the work done to rewrite `<big-term>` will have been wasted.

ACL2 does include one feature that departs from this strict inside-out procedure; calls to `if` are treated specially. The "test" (or "condition") of the `if` is rewritten first, and if it rewrites to a constant, only the appropriate branch of the `if` is rewritten, according to whether the test rewrote to `nil` or true (non-`nil`).

In contrast to ACL2, the Axe Rewriter supports both inside-out and outside-in rewriting. When a function call is being rewritten, outside-in rules are first tried. If any of them fires, the resulting term replaces the original term and is then rewritten, recursively. Only if no outside-in rules apply are the arguments rewritten. Finally, inside-out rules are applied.

If outside-in rewriting were applied to `(car (cons x <big-term>))`, the application of `car` would be rewritten first, by rule `car-cons-fact`, giving just `x`. Since `<big-term>` would be discarded before being rewritten, all the work of rewriting it would be avoided.

Outside-in rewriting can be used to mimic ACL2's special rewriting treatment for `if`, using the following rules:

```
(defthm if-when-not-nil
  (implies test
           (equal (if test x y)
                  x)))


(defthm if-when-nil
  (implies (not test)
           (equal (if test x y)
                  y)))
```

If these rules are in use as outside-in rules, then whenever a call to `if` is rewritten, its test will first be rewritten fully (using all rules, until it stabilizes). If the result is a constant other than `nil`, the rule `if-when-not-nil` will save rewriting the else-branch. Likewise, `if-when-nil` will save rewriting the then-branch if the test rewrites to `nil`. If neither of these rules apply, the arguments of the `if` are rewritten as normal and then any inside-out rules are applied to the call of `if`. One might object that this scheme will cause the `if` test to be rewritten several times (once in the attempt to relieve the hypothesis of `if-when-not-nil`, once for the hypothesis of `if-when-nil`, and again when the arguments to the `if` are rewritten). However, the Axe Rewriter can often memoize the rewrites it performs (see Section 5.16).

Outside-in rules can also be used to implement "short-circuit rewriting" for `booland`, `boolor`, and similar operators. For example, when rewriting a call to `booland`, an outside-in rule can cause the first argument to be rewritten, and, if the result is `nil`, can then replace the whole `booland` term with `nil`. The saves the expense of rewriting the second argument, which would normally be done before the rewriter considers the `booland`. Using outside-in rewriting for such things is simpler, more extensible, and less error prone (because the rules are proved correct) than hard coding equivalent transformations directly into the rewriter.

5.13 Objective-Based Rewriting

The Axe Rewriter can be called with a "rewrite objective" instructing it to attempt to either strengthen or weaken the claim being rewritten, using the known assumptions. For example, consider the terms `(< x 2)` and `(< x 3)`. These terms are not equivalent (they disagree, for example, when `x` is `2`). However, if we have the two assumptions `(integerp x)` and `(not (equal x 2))`, then `(< x 2)` and `(< x 3)` are equivalent for all values of `x` that satisfy the assumptions. In such a situation, it would be sound to rewrite `(< x 2)` to `(< x 3)`, or vice versa. Which of the two terms is preferable depends on the objective provided to the Axe Rewriter. If the objective is to strengthen the term, then `(< x 2)` is preferable because it is the stronger term. If the objective is to weaken the term, then `(< x 3)` is preferable.

When rewriting facts that are to be assumed for a given proof, it is often desirable to strengthen them, so that they provide more information. On the other hand, when rewriting terms that are to be proven, it is often desirable to weaken them, so that they are easier to prove. For example, a hypothesis of a conditional rewrite rule can be rewritten with the objective of weakening it.

The Axe Rewriter supports three rewrite objectives, `t`, `nil`, and `?`. An objective of `t` indicates that the term should be weakened. That is, it should should be transformed in the direction of `t`, which is the weakest term in the ACL2 logic. An objective of `nil` means that the term should be strengthened; `nil` is the strongest term in the ACL2 logic, since an assumption of `nil` allows one to prove anything. An objective of `?` indicates that neither weakening or strengthening should be attempted.

Rewrite rules used by the Axe Rewriter can query the rewrite objective using a mechanism similar to `axe-syntaxp` (Section 5.5). When the special unary function `axe-rewrite-objective` appears as the hypothesis of a rule, the Axe Rewriter checks whether the argument supplied to `axe-rewrite-objective` is the current rewrite objective. If not, the hypothesis fails to be relieved. Like `syntaxp`, `axe-rewrite-objective` is logically just `t`, so it is always sound to consider such a hypothesis to be relieved.

The following rules use `axe-rewrite-objective` to rewrite `(< x 2)` to `(< x 3)`, or vice versa, according to the rewrite objective supplied to the rewriter.

```
(defthm rule1
  (implies (and (axe-rewrite-objective 't)
                (not (equal 2 x))
                (integerp x))
           (equal (< x 2)
                  (< x 3))))


(defthm rule2
  (implies (and (axe-rewrite-objective 'nil)
                (not (equal 2 x))
                (integerp x))
           (equal (< x 3)
                  (< x 2))))
```

Of course, the actual rules used would not have the constants 2 and 3 "hard-coded" into them. These very specific rules are included only for simplicity of explanation.

As the rewriter operates, it tracks the current rewrite objective, and when a call of `not` is encountered, the rewrite objective is inverted when rewriting the argument of the `not`. (Inversion means that `t` becomes `nil`, `nil` becomes `t`, and `?` is unchanged.) Inverting the rewrite objective is appropriate, because to strengthen a call of `not`, one should weaken its argument, and vice versa.

When memoizing (see Section 5.16), the Axe Rewriter avoids rewriting a term more than once, but the use of rewrite objectives provides an exception to that rule. When rewrite objectives are used, a term can in fact be rewritten three times, once with each possible rewrite objective. However, the rewrite objective only makes sense for terms which are intended to be used in a boolean context. Terms of other types (such as calls to functions that return bit-vectors) are always rewritten with an objective of `?`, and these make up the majority of terms in proofs about unrolled cryptographic programs.

In addition to its use for the hypotheses of conditional rules, objective-based rewriting is used after Axe splits a miter into two cases (as described in Section 8.4). After the split, a new fact (the split fact) is known, and rewriting is performed to try to strengthen the existing assumptions using the new fact. Having assumptions that are individually as strong as possible helps when simplifying the miter. The strengthening is especially helpful in the big proofs about SHA-1 and MD5 on messages of unknown length.

5.14 Rule Sets

It is often desirable to apply several different sets of rewrite rules, with the DAG being fully rewritten under each set of rules before the next set is used. For example, miters in Axe can be simplified first at the word level, using one set of rules, then bit-blasted (using special-purpose rewrite rules), then simplified again using bit-level rules.

ACL2 includes a facility for "priority-based rewriting" [4] which implements such a scheme using "computed hints." There are no "hints" in Axe. Instead, Axe allows several rule sets to be passed explicitly to the Rewriter. This causes the rules in the first set to be applied until the term "stabilizes." Then the second set is applied until stability is again achieved, and so on. (The Axe Rewriter is always explicitly passed one or more sets of rewrite rules. This contrasts with ACL2, which always uses all rules that have been defined but not "disabled.")

5.15 Rule Ordering

When rewriting a term there may be many rules that could apply to it. For example, there are many rules that rewrite calls to each of Axe's bit-vector operators. In what order should the rules be tried? In ACL2, the order is determined by when the rules were introduced into the logical session, with more recently introduced rules tried first. This approach may be good for interactive proofs; if a lemma is proved on behalf of some theorem it is likely to immediately precede it in the session and so will be tried first. However, this order may not always be beneficial; for example, the first rules introduced in a theory are often more fundamental, but they will be tried last[13]. (Another problem with ACL2's scheme is that it ties the ordering of rules to the order in which the files that contain them were included in the session, which can be rather arbitrary.)

Axe takes a different approach, giving users total control over the order in which rules are tried. In particular, each rule is assigned a rational number priority. The default priority of a rule is 0, and the user can explicitly assign a different priority to any rule. Rules are tried according to the numerical order of their priorities. For example, a rule with a priority of 2 is tried before a rule with a priority of 3. Rules with negative priorities are tried before any rules with the default priority of 0, and rules with positive numbers are tried last. (The ordering of rules with the same priority is arbitrary.) Since the priorities are rational numbers, there is always room to insert a new priority level between any two existing priority levels.

[13] In order to force an old rule to be tried first, one can define a new rule that is identical to the old rule (but with a different name) and then disable the old rule. Such a scheme seems inelegant.

5.16 Memoization

ACL2's rewriter doesn't perform memoization of the results that it computes, so it may rewrite the same term over and over. In contrast, the Axe Rewriter uses memoization. Unless looping rewrites are present, the rewriting process for any subterm (represented by a node in the DAG) terminates when a "stable" term is reached to which no further rewrites apply. When this happens, the Rewriter updates its internal array of results (indexed by node number) to reflect that the given term rewrote to the corresponding stable term. If the same term is later encountered again (under the same top-level call to the Rewriter), it is immediately replaced with the previously-computed stable term. As mentioned in Section 5.13, the memoization process is sensitive to the rewrite objective used for the term. Thus, each subterm is rewritten at most three times, one for each possible rewrite objective (`t`, `nil`, or `?`).

One disadvantage of memoizing rewrites is that it is incompatible with the use of contextual information provided by `if` nodes above the term being rewritten. It would be unsound to reuse the result of a rewrite done in one context when later rewriting in a different context. For example, rewriting a term of the form:

```
(if <test> <then-branch> <else-branch>)
```

involves first descending into `<then-branch>`, where it is sound to assume `<test>` for all rewriting done. (In any situation for which `<test>` is false, `<then-branch>` is irrelevant.) Likewise, it is sound to assume the negation of `test` when rewriting `<else-branch>`. This is what the ACL2 rewriter does. However, if `<then-branch>` rewrites to some new term A, it would be unsound to memoize this result and later simply replace `<then-branch>` with A, because `<test>` might not hold.

The inability to use contextual information while memoizing is not as bad as it might seem, because contextual information is usually to expensive to use anyway. This is because the number of contexts for a given DAG node may be exponential in the number of `if` nodes above it in the DAG. Each context corresponds to one path from the root of the DAG to the node and represents a choice between the then-branch and the else-branch of each `if` along the path. Since there may be exponentially many such paths if the DAG contains massive sharing, there may be many different contexts in which to rewrite the node. Using them all would require too much time rewriting and might reduce the sharing present the DAG (if the term rewrote to several different new terms in different contexts). Thus, the Axe Rewriter doesn't use contextual information, which allows it to perform memoization[14].

5.17 Relieving Hypotheses with the Axe Prover

It sometimes happens that a hypothesis of a rewrite rule fails to be relieved through normal rewriting but could actually be relieved using other techniques. For example, perhaps splitting into cases or calling the STP decision procedure would suffice to relieve the hypothesis. To ensure that the rewriting process is fast, none of these strategies is normally attempted when relieving a hypothesis. However, the Axe Rewriter includes a facility to indicate that extra effort should be expended to relieve a particular hypothesis of a rule. This is indicated by wrapping the hypothesis in a call to the special unary function `work-hard`. The `work-hard` function is logically just the identity function, so wrapping a hypothesis in `work-hard` has no effect on the meaning of the hypothesis[15]; it simply indicates that extra effort should be expended. When the Axe Rewriter encounters a hypothesis wrapped in `work-hard`, it first tries to relieve it using normal rewriting. If the result is a a non-`nil` constant, the hypothesis counts as relieved (as normal). If the result is `nil`, the hypothesis fails to be relieved

[14] When the Axe Equivalence Checker is proving two nodes equivalent prior to merging them, it does use an approximation of the context of the node being replaced, as discussed in Section 10.5.1.

[15] This is in contrast this with `axe-syntaxp`, `axe-bind-free`, and `axe-rewrite-objective`, all of which are logically `t`.

(again, as normal). If the rewritten hypothesis is not a constant, the the Axe Prover is called to attempt to prove the rewritten hypothesis. As described in Chapter 7, the Axe Prover uses several techniques, including rewriting (of both the term to be proved and all of the assumptions), splitting into cases, and calling the STP decision procedure.

Figure 5.1: `work-hard Example`

```
(defthm bv-array-read-of-bv-array-write-same-work-hard
  (implies (and (natp index)
                (work-hard (< index len))
                (integerp len))
           (equal (bv-array-read width len index
                                 (bv-array-write width len index val lst))
                  (loghead width val))))
```

A typical example of the use of `work-hard` is for the hypotheses of rewrite rules about array operations. For example, the rule in Figure 5.1 says that reading from array location `index`, after writing `val` to that location, simply returns `val` (perhaps trimmed down to size). This is only true if the index is in bounds. The `work-hard` hypothesis expresses the claim that the index is in bounds and indicates that extra work should be expended if necessary to prove it, so that the rule can apply. In some cases, rewriting alone would be insufficient to prove such facts, especially if the terms for the indices or the array lengths included `if` expressions. Whether to add `work-hard` to a hypothesis is a judgment call and is often based on the experience of the rule failing to fire without it.

The ACL2 system has two facilities that are somewhat similar to `work-hard`, namely `force` and `case-split`. Like `work-hard`, both `force` and `case-split` are unary functions logically equivalent to the identity function and so can be wrapped around hypotheses without changing their meanings. Wrapping a hypothesis in a call to `force` will cause it to be simply assumed, with an obligation recorded that the hypothesis must later be proved. If the hypothesis fails to be proved during the subsequent “forcing round,” the whole proof is invalidated. This is in contrast to `work-hard`, for which failure to prove the hypothesis causes the hypothesis to fail to be relieved but allows the rewriting process to continue.

The function `case-split` is similar to `force` except that it causes the ACL2 “goal” to be split into two cases, one in which the given hypothesis is assumed true (and in which the rewrite rule then applies), and another in which the hypothesis is assumed false. Although splitting into cases in this way makes sense when one is trying to prove that some goal is true (splitting simply adds more goals to prove), it would not really make sense for the Axe Rewriter, which is often merely used to simplify a term, rather than to prove a goal. Perhaps something like `case-split` could be used to cause the Rewriter to generate a call to `if`, but seems undesirable.

For the purposes of Axe, `work-hard` seems superior to `force` or `case-split`. (One benefit of `force` is that if the same fact is `force`d repeatedly, the proof obligations can in some cases be collected and discharged all at once. A similar facility could be added to Axe by memoizing calls to the Axe Prover done on behalf of `work-hard`.)

Chapter 6

Domain Specific Rewrites Used By Axe

Axe contains a large set of rewrite rules about its built-in functions; the rules are applied by the Axe Rewriter, and many of them use the Rewriter's advanced features[1]. Almost all of Axe's rules were proved correct in ACL2[2]. Many of Axe's rewrite rules are general purpose, dealing with common topics such as bit-vectors, arrays, and sequences. Axe includes hundreds of rewrite rules, and it is not possible to describe them all here. This chapter focuses on some of the most innovative rules used by Axe, namely the domain-specific rewrites that it uses to handle common operations and coding idioms in cryptographic programs. (Axe's special handling of XOR operations, described in Section 5.10, is another domain-specific optimization for cryptography.)

[1] The rules are also used by the Axe Prover, discussed in the next chapter.

[2] A few tricky proofs were temporarily skipped; they should be performed if Axe ever transitions from a research prototype to a production-level tool.

6.1 Rewrites for Bit-Vector Rotation

A common operation in cryptographic code is bit-vector rotation. Rotations are similar to shift operations, except that bits shifted past the end of the bit-vector wrap around to the other end (instead of being lost). For example, rotating the 16-bit[3] value

```
1010101000001111
```

to the right by three places gives

```
1111010101000001
```

Rotations are common in cryptographic code because they lose no information and can be reversed by simply rotating in the other direction. Axe contains built-in rotation operations, `leftrotate` and `rightrotate`, that are used in its specifications of cryptographic algorithms. (The `rightrotate` operator is just a convenience, since a right rotation can always be expressed as an equivalent left rotation.)

Unfortunately, Java bytecode lacks native instructions for bit-vector rotation. Thus, programmers must implement the rotation using a sequence of other operations. However, there is more than one equivalent coding idiom for doing so, and different choices made by different programmers complicate the process of equivalence checking. For this reason, Axe includes several rules to normalize such equivalent expressions. These techniques were described, along with an early version of Axe, in a 2008 paper by the author of this dissertation and his advisor [58].

The common idiom to perform a rotation (in the absence of a dedicated rotate operator), is to perform two shifts in opposite directions and then combine the results. For example, the rotation above can be computed by first shifting to the right by three places, giving

[3] The JVM usually operates on 32-bit quantities, but we use 16 bits here for brevity.

```
0001010101000001
```

then shifting to the left by 13 places (the difference of the field width and the rotation amount), giving

```
1110000000000000
```

and finally combining the results using OR, as follows:

```
   0001010101000001
OR 1110000000000000
-------------------
   1111010101000001
```

Note that the individual shift operations introduce zeros into the bit positions opened by the shifts. This always occurs in such a way that the OR operation never ORs two ones in the same position. Thus, the combination step could instead be performed using XOR (which agrees with OR for each bit position not involving two ones). Bit-vector addition could also be used; the lack of two ones in any position will prevent any carry bits from being propagated. The possibility of different but functionally equivalent coding idioms must be dealt with during equivalence checking.

When the rotation amount is a constant (e.g., 3 in this example), the left and right shift amounts will also be constant (3 and 13 in the example), and the rotation can be handled by bit-blasting; the rotation operation can be expressed as a particular concatenation of the individual bits of the input, because the rotation amount determines where every bit of the input will end up in the output. However, “variable rotations” are also possible, in which the rotation amount is not constant but depends on the input to the program. One of the distinguishing characteristics of the RC6 block cipher is its uses of such “data-dependent rotations.”

Variable rotations are difficult to handle, because, depending on the unknown rotation amount, any bit of the input could end up in any bit position in the output.

It is not possible to directly bit-blast a variable rotation. One possibility would be to generate an if-then-else nest with one branch for each possible rotation amount (32 branches for a rotation in a 32-bit field), but experience with Axe has shown this to be unwieldy. A better approach is to handle variable rotations by rewriting them to a normal form, specifically, Axe's `leftrotate` operator. For example, since the Java code for RC6 uses OR to combine the results of the two shift operations, Axe includes the rewrite rule[4]:

```
(defthm bvor-of-shl-and-shr-becomes-leftrotate
  (implies (and (equal size (+ amt amt2))
                (natp amt)
                (natp amt2))
           (equal (bvor size (shl size x amt) (shr size x amt2))
                  (leftrotate size amt x))))
```

This rule recognizes the rotation idiom and replaces it with the nice `leftrotate` operator. Note the first hypothesis, which requires the sum of the rotation amounts to be the bit width of the OR. (In the example above, we had $3 + 13 = 16$.) This hypothesis is not trivial to relieve when `amt` and `amt2` are not constants, but Axe contains sufficient rewrite rules (about bit-vector subtraction, etc.) to enable it to be rewritten to `t` in the RC6 proof.

Without rewrites to normalize the rotation operations, verification of RC6 could be difficult. (RC6 also includes 32-bit multiplication operations, and SAT-based tools perform poorly on examples that include multiplication.) However, with the rules in Axe's library, the formal proof about RC6 encryption takes less than 12 seconds (including the symbolic execution of the bytecode). The proof succeeds using rewriting alone (with no test cases or sweeping and merging required).

[4] Actually, several variants of the rule are included, for example, a version with the operands to `bvor` commuted into the other order.

6.2 Handling Patterns in Lookup Tables

Cryptographic code is often performance-critical, and real implementations can be heavily optimized. One common optimization is to precompute values, when possible, and store the results in lookup tables. This could allow a sequence of logical operations (XORs, bit extractions, concatenations, etc.) to be replaced by a quick table lookup. The presence of such lookup tables in optimized implementations complicates equivalence checking when comparison must be made to unoptimized reference implementations or formal specifications.

In Axe, lookup tables are represented as constant arrays. Axe contains rewrite rules that examine the values in such arrays, and, based on patterns present in them, can sometimes turn table accesses back into the equivalent sequences of logical operations.

Consider a three-bit bit-vector `v` whose bits can be represented as $v_2v_1v_0$. Imagine that we wish to calculate the XOR of each pair of bits and form a vector of the results. The result (call it `R`) will contain three bits and might be expressed as:

(v_2 `XOR` v_1) `@` (v_2 `XOR` v_0) `@` (v_1 `XOR` v_0)

Here, `@` denotes bit-vector concatenation. `R` could be calculated on the JVM using a series of logical operations, but the process would be a bit complicated. To compute each bit of `R`, one would need to perform some combination of masking out the individual bits to be operated on, shifting them into position, and XORing them together. One would then create `R` by combining the results for its individual bits. Instead of doing all this, one could simply use the following lookup table, `T` (with indices and table entries expressed in binary[5]):

```
T[000] = 00000000
T[001] = 00000011
```

[5] The table entries are 8-bits wide because in Java they would be `bytes`.

```
T[010] = 00000101
T[011] = 00000110
T[100] = 00000110
T[101] = 00000101
T[110] = 00000011
T[111] = 00000000
```

To compute the correct value of all three bits of `R`, one can just access `T` at index $v_2v_1v_0$. This sort of thing is typical of what happens in block cipher code, with the values in lookup tables representing XORs of the tables' index bits. Note that there are certain patterns in the table values, and Axe's rewrite rules can use these patterns to convert table lookups back into logical formulas.

The first step is to bit-blast `T`, yielding eight new tables, `T7` down through `T0`, corresponding to the individual bits of the data values in `T`. Each of the resulting tables will contain single-bit values and will correspond to a "column" of bits from `T`. For example, `T0`, representing the least significant bits of `T`, will be:

```
T0[000] = 0
T0[001] = 1
T0[010] = 1
T0[011] = 0
T0[100] = 0
T0[101] = 1
T0[110] = 1
T0[111] = 0
```

Bit-blasting replaces the table access `T[v]` with the equivalent expression

```
T7[v] @ T6[v] @ T5[v] @ T4[v] @ T3[v] @ T2[v] @ T1[v] @ T0[v]
```

Axe then uses rewrite rules to simplify this expression. First, note that the first five tables (T7 down through T3) will contain all zeros. Axe can rewrite the lookups into those tables to 0, using a rule that applies whenever all of an array's elements are the same. (The rule applies only to arrays that are quoted constants, for which the equality of all elements can be quickly checked.)

The remaining tables, T2, T1, and T0, have values that do depend on their index bits; each value is the XOR of some subset of the index bits. For example, the value of T0[v] is the XOR of the low two bits of v. Axe can turn such tables back into logical operations, using rewrite rules that detect patterns in the tables. Note that the first half of the values in T0 (for indices 000 through 011) is the same as the second half (for indices 100 through 111). This indicates that the top-most index bit is irrelevant. Axe contains a rewrite rule, array-reduction-when-top-bit-is-irrelevant, to transform such an array access into an equivalent access that uses an index that smaller by one bit and an array of half the size. The rule is similar to array-reduction-when-all-same but is more complicated, since it requires cutting the array in half[6]. The application of the rule would replace the lookup into T0 with an equivalent lookup, using one less index bit, and using a shorter table (call it T0new):

```
T0new[00] = 0
T0new[01] = 1
T0new[10] = 1
T0new[11] = 0
```

T0new also contains a pattern: the bits in the first half of the table are the complements of the bits in the second half of the table. This indicates that the top-most index bit is "XORed in" to the table. That is, an access into T0new with index v_1v_0

[6] The original array length must be a power of 2, and the new index width will be ones less than the base 2 logarithm of that value.

can be replaced by the XOR of v_1 and an access into a smaller table (call it `T0new2`) using the remaining index v_0. `T0new2` would simply be:

```
T0new2[0]=0
T0new2[1]=1
```

In essence, the high bit v_1 simply indicates whether the value read from `T0new2` should be complemented.

The transformation just described is expressed as the Axe rewrite rule `array-reduction-when-top-bit-is-xored-in`, which is complicated but is similar to the other rules described in this section.

Finally, it is clear that accessing `T0new2` using index v_0 just returns v_0, and Axe has a rewrite rule, `array-reduction-0-1`, to express that fact. All of the above rules are applied automatically by Axe (with appropriate use of `syntaxp` to keep them from being too expensive for non-constant-arrays), and the cumulative effect is to rewrite the access of `T0` at index $v_2 v_1 v_0$ into simply v_1 `XOR` v_0. Similar chains of simplifications are applied to rewrite the accesses into tables `T1` and `T2`, and the final result expresses the lookup into the original table T as:
`(`v_2 `XOR` v_1`) @ (`v_2 `XOR` v_0`) @ (`v_1 `XOR` v_0`)` This is the logical expression that was originally used to generate the values of T during optimization.

The automatic application of the transformations in this section helps Axe to verify optimized real-world implementations of block ciphers, including AES.

6.3 Rewrites for Data Packing

A key operation in cryptographic code is data packing, in which small data elements (such as bytes) are combined into larger data elements (such as 32-bit words). As described in Section 4.10.8, Axe contains built-in operators for data packing and unpacking. It also contains rewrite rules to reason about such operations. The ability to express packing using built-in functions that Axe knows how to reason about is critical to the successful application of Axe to verify cryptographic hash functions.

For computations that involve packing a finite amount of data, the packing operations can often be expanded away (e.g., into concatenations). The real benefit arises when the amount of data to be packed is not bounded. Such operations occur in the SHA-1 and MD5 hash functions, each of which takes a message of unknown length, pads it, packs the padded message (by packing the bytes into words and the words into blocks), and then processes the blocks[7]. Such a packing operation cannot be represented using a finite number of standard bit-vector operations, since the amount of data to be processed is unknown. Some sort of loop or recursion is needed; as we will see, Axe's built in operations (such as `map-packbv`) come in very handy.

A major issue in the verification of the Bouncy Castle implementation of SHA-1 is that it does not perform the dictated packing all at once (before processing any blocks, as indicated by the specification). Instead, the packing is integrated with the processing of each block. Thus, two separate computations in the specification (packing into blocks and processing the blocks) correspond to a single loop in the implementation. A critical issue is how to represent the packing operation in the formal specification of SHA-1. One could simply write a custom recursive function to perform the packing. However, the equivalence proof will depend on some critical properties of that packing operation. In particular, when data is read out of the packed blocks, it must be the correct data (so that the specification can be compared to the implementation that does the packing just-in-time).

The SHA-1 proof involves reasoning about concepts like "the nth element of the packed data," which can be expressed (in a complicated way) in terms of the bytes

[7] This discussion will focus on SHA-1; MD5 is slightly more complicated in that the order of the bytes in each group of four is reversed before they are packed into a word (giving the packing a different "endianness").

passed in to the packing operation. However, automatically discovering and proving such facts for the custom SHA-1 packing function would be tricky.

Instead of a custom recursive function, Axe's built-in packing functions can be used in the specification. In particular, the packing for SHA-1 is essentially:

```
(group '16 (map-packbv '4 '8 (group '4 bytes)))
```

This says that we take the bytes, group them into groups of 4, pack each group into a 32-bit word, and then group the words into blocks of 16. The advantage of this formulation is that `group` and `map-packbv` are built-in to Axe, and Axe contains rewrite rules about them. In particular, Axe has rules that can simplify the extraction of the nth element of this packing expression (yielding the expression for the correct block of data), and to simplify the extraction of the nth element of that block (yielding an expression for a particular word of data). The details are quite complicated, but they use lemmas such as:

```
(defthm nth-of-map-packbv
  (implies (natp n)
           (equal (nth n (map-packbv itemcount itemsize items-lst))
                  (if (< n (len items-lst))
                      (packbv itemcount itemsize (nth n items-lst))
                      'nil))))
```

This lemma was automatically generated by `defmap` (described in Section 4.10.7) when it was used to define `map-packbv`. It expresses how to push an `nth` through a call to `map-packbv`. (If we pack several items and then extract the nth element of the result, that is the same as first taking the nth element and then packing it.)

The key advantage of Axe's packing functions is that such lemmas already exist for them, and those lemmas can be automatically applied. (Of course, the packing functions also simplify the creation of formal specifications.) Finally, the packing operations are general enough (operating on data elements of arbitrary size) to be re-used in several specifications.

Chapter 7

The Axe Prover

Axe includes a theorem prover which can apply a variety of techniques, including rewriting, substitution, "tuple elimination," splitting into cases, and querying the STP decision procedure. Throughout its operation, the Axe Prover uses the DAG representation of terms. It takes as input a DAG and a set of node numbers in that DAG, and it attempts to prove the disjunction of the terms represented by the indicated nodes. That is, it tries to show that the disjunction evaluates to true, for all possible values of the variables. Like the Axe Rewriter, the Axe Prover is unrestricted in the operators it can accept; all `:logic` mode ACL2 functions are allowed. The Axe Prover is designed to be sound with respect to the ACL2 logic, but it is necessarily incomplete, since no decision procedure exists for arbitrary operators. However, because STP is complete, the Axe Prover is also complete for those goals which can be accurately represented as STP queries[1]. As described below, operators that cannot be represented in STP are "cut" out of the proof before STP is called, but this may introduce incompleteness.

The Axe Prover is used in several different ways by the rest of the Axe system, including to prove two nodes equal prior to merging them (Section 8.3.3), to discharge

[1] In practice, STP may timeout or run out of memory on large goals.

a `work-hard` hypothesis in the Axe Rewriter (Section 5.17), to prove claims about the return values of loop functions (Section 11.9), and to prove that the exit tests of two loops are equivalent (Section 12.1). The Axe Prover can also be called as a standalone tool or be called during an ACL2 proof (via a `:clause-processor` hint).

The Axe Prover is similar to the Axe Rewriter but applies more powerful reasoning. While the Rewriter can sometimes prove a claim by rewriting it to true (given a set of assumptions), it never rewrites those assumptions, even though doing so may be necessary to complete the proof (e.g., if the assumptions are not in simplified form). Nor does the Rewriter split into cases if the assumptions contain `if`s. In contrast, the Axe Prover treats all disjuncts in the goal equally, rewriting each one using information provided by the others. The Prover also applies techniques not used by the Rewriter (splitting into cases, calling STP, etc.)[2].

7.1 The Clause Form

The Axe Prover represents goals as disjunctions, which it tries to prove are true for all values of their variables. Such a disjunction is called a "clause."

Consider a claim of the form:

```
(implies (and H1 H2 ... Hn) C)
```

where the `H` terms are the hypotheses and `C` is the conclusion. Such a claim can be represented as the disjunction

```
(or (not H1) (not H2) ... (not Hn) C)
```

and it is upon this disjunction that the Prover would operate.

[2] Technically the Rewriter can do these things, but only by calling the Prover (to discharge a `work-hard` hypothesis).

When proving a disjunction, it suffices to transform any of the disjuncts into true. That is, the Prover can establish the truth of the conclusion or the falsity of any hypothesis (which would establish the truth of the negated hypothesis and thus prove the clause). The Axe Prover treats all disjuncts of the clause (conclusion and negated hypotheses) equally.

The Axe Prover operates in several phases. If any phase proves the goal, the proof succeeds. Otherwise, the next phase is tried. First the Prover rewrites the clause until stability is reached. Then substitution and tuple elimination are performed. Next, STP is called (after possibly cutting out certain nodes of the DAG). Finally, the Prover may split the goal into two cases, based on some boolean-valued subterm, and then try to prove each case[3].

7.2 Proof Via Rewriting

The Axe Prover begins by rewriting the disjuncts in the clause. If any disjunct rewrites to a "true" (non-`nil`) constant, that disjunct suffices to prove the clause, because a disjunction is true if any of its disjuncts is true. If any disjunct rewrites to the constant `nil`, it is dropped, since a disjunct of "false" can be soundly dropped from a disjunction. If a disjunct rewrites to a non-constant term, it is replaced in the clause with that simplified term, and the rewriting process proceeds to the next disjunct.

When rewriting a disjunct, the Prover assumes the negations of all of the other disjuncts. This is sound because if such an assumption is wrong, then some other disjunct must actually be true. But the truth of that disjunct would then suffice to prove the clause.

[3] To prevent excessively expensive proofs, splitting is only attempted if calling STP did not timeout. The timeout amount is passed in to the prover and by default is three seconds. Facilities exist outside the prover for increasing this timeout in certain cases.

Processing each disjunct in turn, replacing each with its rewritten form, constitutes one "pass" through the disjuncts. During a pass the Prover tracks whether any disjunct has changed. If anything changes on a pass, another pass is performed, until eventually nothing changes. At that point, the clause is said to be "stable" under rewriting. If a bad set of rewrite rules is used, the rewriting process may enter an infinite loop, either when rewriting a single disjunct, or because the clause never becomes stable. Some care has been taken to prevent such situations when using the set of rewrite rules provided by Axe.

If a disjunct rewrites to a disjunction, the Axe Prover performs "promotion" of the disjuncts. For example, if a disjunct rewrites to `(boolor x y)`[4], then instead of replacing the disjunct with the new term `(boolor x y)`, Axe replaces it with both `x` and `y`, which thus become top-level disjuncts in the clause. In fact, a more general mechanism is used to harvest new disjuncts from the boolean structure of a rewritten disjunct. In addition to calls to `boolor`, that mechanism can harvest disjuncts from terms like `(not (booland x y))`, whose disjuncts are `(not x)` and `(not y)`. The algorithm works as follows:

- The disjuncts of `(boolor x y)` are the union of the disjuncts of `x` and the disjuncts of `y`.
- The disjuncts of `(not x)` are the negations of the conjuncts of `x`.
- The conjuncts of `(booland x y)` are the union of the conjuncts of `x` and the conjuncts of `y`.
- The conjuncts of `(not x)` are the negations of the disjuncts of `x`.
- The disjuncts or conjuncts of any other term are the singleton set containing that term.

[4] Of course, the Prover actually represents terms in DAG form, but for clarity we show them as ACL2 terms throughout this chapter.

The rewriting done to a disjunct incorporates many of the advanced rewriting features discussed in Chapter 5, such free variable matching, `axe-syntaxp`, and `axe-bind-free`.

Rewriting a disjunct does not change any of the other disjuncts in the clause, even if they share DAG nodes with the disjunct being rewritten. When a disjunct is changed during rewriting, any new nodes that arise are added to the DAG, not changed in place.

If after a rewriting pass, no disjuncts remain, the proof fails. (The disjunction of no disjuncts is “false”.)

7.3 Substitution

Performing substitution using known equalities can be a helpful strategy for a theorem prover. However, it can give rise to tricky issues about which way each equality should be directed and about how to avoid looping behavior. While more sophisticated schemes are possible, the Axe Prover is fairly conservative in its approach.

When proving a clause, a disjunct of the form `(not (equal <x> <y>))` provides a known equality that can be assumed when rewriting the other disjuncts. (If `<x>` and `<y>` are not in fact equal, then the entire clause is true.) Substitution using such an equality is done by the Axe Prover in only two situations. First, the Prover always replaces a term with a constant to which it is equal. This occurs during the rewriting process and seems safe, since a constant is maximally simplified.

Second, the Prover uses equalities to eliminate variables. After a clause is stable under rewriting, the Prover looks for disjuncts of the form `(not (equal <variable> <term>))` or `(not (equal <term> <variable>))`. To prevent loops, the Prover abstains from substitution if `<term>` mentions `<variable>`. Otherwise, the Prover uses the equality by replacing `<variable>` with `<term>` in all other disjuncts. Finally, it removed the negated equality from the set of disjuncts. If the resulting clause is valid, then so is the original clause.

Axe's conservative approach to equality substitution suffices for the proofs discussed in this dissertation.

7.4 Tuple Elimination

Many of the values dealt with by Axe are tuples. For example, a loop function may take a single argument, `params`, that represents all of its loop variables. (Upon exit, it would simply return `params`.) In such a case, each component of the tuple, such as `(nth '2 params)`, is essentially a separate variable, and substituting to eliminate that "variable" might be desirable. However, a term such as `(nth '2 params)` is not syntactically a variable. Axe addresses this issue by performing "tuple elimination," in which a tuple variable is replaced with a `cons` nest containing fresh variables corresponding to the components of the tuple. For example, if `params` is a tuple of length 3, it would be replaced everywhere it appears with `(cons params0 (cons params1 (cons params2 'nil)))` where `params0`, `params1`, and `params2` are fresh variables. It is only correct to make such a substitution if the tuple variable is known to have a length that is a particular constant (3 in this example) and is known to be a true list. These facts are found by examining the disjuncts in the clause. (For example, a disjunct of `(not (true-listp params))` indicates that `params` can safely be assumed to be a true list.)

Once tuple elimination is performed, variable substitution may be performed on the resulting fresh variables. The Axe Prover's tuple elimination operation is related to ACL2's "destructor elimination." (Of course, the process in Axe operates on DAGs.)

During substitution and tuple elimination, the Axe Prover tracks whether any changes were made to the clause. If so, the proof process starts over on the resulting clause, beginning with rewriting.

7.5 Calling STP

If rewriting, substitution, and tuple elimination are not sufficient to prove the clause, the Axe Prover calls the external decision procedure STP. STP can decide queries that contain bit-vector, array, and boolean operators. Because the Axe Prover supports arbitrary operators, some portions of the goal may not be representable in STP's language. Such subterms are "cut" out of the DAG; that is, their DAG nodes (called "cut nodes") are replaced with fresh variables in the STP query. The result is a more general goal. If STP reports that the cut goal is valid, then so is the original goal. (Validity of the cut goal means it is valid for all instantiations of the cut variables, including the particular instantiation represented by the terms cut out.)

If STP fails to prove the cut goal, by reporting that it is invalid or by timing out, the call to STP tells us nothing definitive about the original goal. In particular, if STP says that the cut goal is invalid, there must be some counterexample (a valuation for the cut nodes) that makes the cut goal false. However, this counterexample may be a "false negative." That is, it may not actually be a possible valuation for the cut nodes, if one considers the full, uncut goal. If the portion of the DAG below the cut prevents all possible counterexamples of the cut goal, the uncut goal may in fact be valid even though the cut goal is invalid. For example, consider the valid goal `(equal (bvxor '32 x y) (bvxor '32 y x))`. If one cuts out the `bvxor` nodes and calls STP on the cut goal `(equal cutnode1 cutnode2)`, STP will report that it is invalid. Any valuation for which `cutnode1` and `cutnode2` differ is a counterexample to the cut goal. However, because `bvxor` is commutative, the arguments to `equal` in the uncut goal must always be equal, and all counterexamples to the cut goal are false negatives.

Cutting is also used as a heuristic to improve the performance of STP by cutting out large, irrelevant terms, as described in Section 7.6.

The soundness guarantee when calling STP on a goal is the same as the guarantee for the Axe Prover as a whole; if STP reports that a query is valid, then the original goal that led to the query should be true in the ACL2 logic, for all values of its variables. Of course, this means that soundness of the Axe thus depends on the soundness of STP. Because important structure may be cut out before generating the STP query, this process is clearly not complete. However, if nothing is cut out (because the original goal only includes bit-vector, boolean, and array operations accepted by STP), then completeness follows from the completeness of STP.

The process of calling STP on a goal can be divided into two steps. First, the Axe Prover selects the cut nodes. This process also involves determining the types of the cut nodes, since they will correspond to variables in the STP query and STP variables must be declared with explicit types. The second step is to actually translate the cut goal into STP's input language. This step is complicated by differences, in logic and operators, between Axe and STP.

The next two sections describe in detail how Axe calls STP on a goal. The second step (generating the query given the cut nodes and their types) is discussed first, followed by discussions of how cutting is done to handle non-translatable operators and to heuristically avoid sending irrelevant terms to STP.

In addition to being called from the Axe Prover, STP is also called by the Axe Equivalence Checker to prove the equality of nodes in a "pure" miter (one whose operators can all be represented in STP). In that case, the translation process is essentially the process described in this section, except that no complicated type inference is necessary and the cutting heuristic is slightly different. (See Section 8.3.3.)

7.5.1 Translating To STP Given The Cuts and Types

Given a set of cut nodes and their types, turning an Axe Prover goal into an STP query is fairly straightforward, except for complications due to incompatibilities between the operators and the logics of STP and Axe.

STP is stricter than Axe in that not all terms are "legal." First, STP requires that the widths of all bit-vector operations be fixed. For example, the 32-bit exclusive-or operation `(bvxor '32 x y)` can be expressed in STP, but the more general operation `(bvxor size x y)` cannot, since `size` is not a constant[5]. Likewise, bit indices must be constants, so `(slice '7 '5 x)` is expressible, but `(slice high low x)` is not. Also, array operations must have lengths and bit widths that are constants. (STP further requires that array lengths be powers of 2; Section 7.5.1 describes the translation of arrays of other lengths.)

Furthermore, STP expressions must type check; that is, operators must be applied to operands of the expected type. For example, a bit-vector or array operator should not take a boolean operand[6]. Also, the sizes of bit-vector operands must be consistent. For example, two values being XORed must have the same bit width. Also, even though it is legal in Axe to call `equal` on operands of different types (in which case `equal` returns false), an equality in STP must be between values of the same type.

Throughout this section we assume that all nodes to be translated are in fact expressible somehow in STP. Nodes that are not expressible should be cut out before translation is attempted.

Axe sends queries to STP by writing them to files. This is probably not as fast as interfacing with STP as a library in memory, but the overhead of reading and writing

[5] The latter is perfectly acceptable in ACL2; indeed theorems above bit-vector operators are usually proved for arbitrary sizes.

[6] Booleans could be made compatible by with bit-vectors by representing them as single bits, but in both Axe and STP "true" and "false" are distinct from the bit-vector values 1 and 0.

files is not the bottleneck in Axe's performance[7]. One advantage of writing queries to files is that the files can be easily inspected for debugging purposes.

The query file sent to STP has several parts: declarations of the types of the cut nodes, declarations and assertions for constant arrays (as described below), and the query itself.

As previously described, a goal in the Axe Prover is represented as a disjunction of nodes in some over-arching DAG. Because there may be extensive sharing both within each disjunct and between the disjuncts, it would be inefficient to generate STP queries as trees. In order to compactly represent all of the relevant DAG nodes, Axe generates a large nested LET expression that corresponds to the DAG being translated. A variable is introduced for each relevant DAG node and is bound in the LET to the translation of its expression in terms of other nodes.

For example, the simple formula:

```
(equal (bvxor '32 x y) (bvxor '32 y x))
```

expresses the commutativity of the exclusive-or operation applied to 32-bit arguments. The corresponding STP query is:

```
NODE1 : BITVECTOR(32);
NODE0 : BITVECTOR(32);
QUERY (
LET NODE2 = (BVXOR(NODE0[31:0],NODE1[31:0])) IN (
LET NODE3 = (BVXOR(NODE1[31:0],NODE0[31:0])) IN (
LET NODE4 = (NODE2 = NODE3) IN (
(NODE4)))));
```

This is best understood by comparing it to the DAG representation of the original query:

[7] The files are written into the /tmp/ directory to ensure that they are reside on the local machine and to avoid the overhead of a networked file system.

```
((4 equal 2 3)
 (3 bvxor '32 1 0)
 (2 bvxor '32 0 1)
 (1 . y)
 (0 . x))
```

Each variable in the query corresponds to the DAG node with the same number. The query variables `NODE0` and `NODE1` are declared to be bit-vectors of width 32 and correspond to the variables `x` and `y` at DAG nodes 0 and 1. The variables `NODE2` and `NODE3` are bound in `LET`s to expressions that correspond to the `bvxor` expressions in the DAG. Finally, node 4, the equality, is bound to a call of `=` in a `LET` expression. After all the LETs comes the core of the query, which is simply `(NODE4)`. That is, the query asks whether `NODE4` is always true.

This query is valid, because of the commutativity of `bvxor`, and STP proves it almost immediately[8]. All of the operations in the query are translatable to STP, so no structure is cut out of the DAG. (We assume for this discussion that heuristic cuts are disabled.) Also, we have not specified exactly how the types of the variables `NODE0` and `NODE1` were derived. Although obvious for this example, generating types for nodes is not always so clear. Section 7.5.2 discusses these issues.

It remains to discuss how to translate the expression for each DAG node. In general, Axe's boolean, bit-vector, and array operators correspond well with STP's operators. In the example above, the `bvxor` nodes in the DAG were simply translated using STP's `BVXOR` operator, and the translation for many other operators is similarly straightforward. However, there are a few differences between STP's language and the ACL2 logic which complicate the translation of certain operations.

[8] Of course, such a simple query would usually be proved via rewriting, using the commutativity of XOR. (However, there may be cases during the equivalence checking process, after nodes are merged, in which a query this simple can arise.)

One common complication is that Axe's bit-vector and array operators carry explicit sizes, and bit-vector operands which are too large or small are implicitly chopped down or padded, respectively. The translation reflects this; operands that are too large are chopped down to size (by the inclusion of a bracket expression such as `[31:0]`), and operands that are too narrow are padded by including in the translation a concatenation with the appropriate number of zeros. The next few sections describe issues with specific operators.

Equalities and Extensional Arrays

One issue arises with the translation of equalities. In Axe, the operator `equal` can apply to objects of any type. STP uses `<=>` for booleans and `=` for bit-vectors, but it is enough to choose the correct operator when translating. More complicated is the case of `equal` called on operands that are arrays. Such an expression is legal in Axe, but STP lacks an operator for equating arrays. That is, STP's theory of arrays is not "extensional". In this case, Axe translates the equality of two arrays as a conjunction of equalities, one for each element. (As discussed above, the lengths of arrays must be known constants if translation to STP is to be performed.)

Constant Arrays

Another issue exists with constants that are arrays. Consider the claim:

```
(equal i (bv-array-read '1 '2 i '(0 1)))
```

where `i` is known to be a single bit. (Here `'1` is the bit width of each array element and `'2` is the length of the array.) Note that `bv-array-read` is applied to read from a constant array, represented by `'(0 1)`, in which element `0` is `0` and element `1` is `1`. The formula claims that the value at index `i` in the array is in fact `i`, for all `i`. Given the assumption that `i` is a single bit, this formula is valid. However, because

STP has no syntax for constant arrays, Axe must handle the translation of `'(0 1)` specially. To do so, Axe creates an array variable (which must be fresh) and uses it in place of the constant array. It then generates constraints that specify the value of each element of the new array variable. The DAG representation of the query above is:

```
((2 EQUAL 1 0)
 (1 BV-ARRAY-READ '1 '2 0 '(0 1))
 (0 . I))
```

and from this Axe generates the STP query:

```
NODE0 : BITVECTOR(1);
ARRAYVAR0 : ARRAY BITVECTOR(1) OF BITVECTOR(1);
ASSERT ARRAYVAR0[0bin1]=0bin1;
ASSERT ARRAYVAR0[0bin0]=0bin0;
QUERY (
LET NODE1 = (ARRAYVAR0[NODE0[0:0]][0:0]) IN (
LET NODE2 = (NODE1 = NODE0) IN (
(NODE2))));
```

Here node 1 is translated as an array lookup into the array `ARRAYVAR0`, which represents the constant `'(0 1)`, and the query includes `ASSERT`s that specify the values of each of its two elements.

Constant arrays are common in proofs about cryptographic code because such algorithms often make use of “s-boxes” or “p-boxes”. When translating constant arrays, Axe reuses array variables and their corresponding constraints if the same constant array is found more than once in a query.

Claims about `unsigned-bytep`

In the example in the previous section, `i` was a single bit. This would be represented as `(unsigned-byte-p '1 i)`. However, the `unsigned-byte-p` term does not actually appear in the STP query. Instead, it is reflected in the declaration that variable `NODE0` has type `BITVECTOR(1)`. It is quite common for type facts in a goal to be used in such a way to generate type declarations in the STP query. However, `unsigned-bytep` claims may also appear in other contexts, such as in the conditions of it-then-else expressions, or as claims that are known to be false, rather than true. For example, `(not (unsigned-byte-p '8 x))` expresses the fact that `x` does not fit in 8 bits (i.e., it must have a 1 somewhere above bit 7). If it is known that `x` fits in, say, 16 bits, `(not (unsigned-byte-p '8 x))` is equivalent to `(not (equal '0 (slice '15 '8 x)))`, and this formulation in terms of `slice` dictates how such a fact is translated to STP.

Rotations

The Axe Prover currently has only limited support for translating bit-vector rotation operations into the language of STP; it only handles the `leftrotate32` operator and requires that the rotation amount be constant. Such a rotation can be translated as the concatenation of a “high slice” and a “low slice,” with the high slice occupying the low bits of the result, and vice versa[9]. It would be straightforward to extend the translation to support bit widths other than 32 (though some care would have to be taken for widths that are not powers of 2).

However, a “variable” rotation, in which the rotation amount is not a constant, cannot be directly translated to STP. This is because STP requires all bit widths and indices to be constants, and the slices used in the translation of a rotation have widths

[9] In the special case of a rotation amount of 0, one of these slices would be empty. In this case, the rotation has no effect, and the translation reflects that fact.

that depend on the rotation amount. One option for handling a variable rotation, though far from ideal, is to use rewrite rules to transform the term into an if-then-else nest with one branch per possible rotation amount. Each branch would then have a constant rotation amount and so could be translated to STP.

It is not necessary to translate the right rotation operators, because they can always be rewritten to equivalent expressions using left rotation.

Arrays With Lengths That Are Not Powers of 2

STP requires arrays to have lengths that are powers of 2, but no such restriction applies in Axe. Thus, when translating a goal involving arrays, Axe rounds the length of each array up to a power of 2, and this larger "underlying array" is used in the translation. For example, an Axe array of length 10 would have an underlying array of length 16. Axe's array operators are designed to have simple behavior on out-of-bounds array accesses, and the translation reflects this. In particular, the translation of an array access includes an if-then-else expression that checks whether the index is within the bounds of true Axe array. The else-branch of the if-then-else reflects the appropriate out-of-bounds behavior: out-of-bounds reads return 0, and out-of-bounds writes have no effect. In the common case of arrays whose lengths are in fact powers of 2 (and for which every bit-vector of the correct width is in fact a valid index), these checks are omitted.

Division and Modulo By 0

The division and remainder operators `bvdiv` and `bvmod` have special behavior when dividing by 0. Their translation is similar to the case of arrays whose lengths are not powers of 2; if-then-else nodes are generated with else-branches that match the Axe operators' behavior in the unusual cases.

7.5.2 Cutting Out Non-Translatable Operators

The previous section describes how Axe translates a goal to STP given the cut nodes and their types. Before that can happen, Axe must decide where to put the cuts.

First consider the problem of cuts that are necessary because not all functions can be translated to STP's language. Consider the following formula:

```
(implies (unsigned-byte-p '1 (len x))
         (boolor (equal '0 (len x))
                 (equal '1 (len x))))
```

This formula is valid (because a bit-vector of width 1 must either be 0 or 1). However, the term `(len x)` cannot be translated to STP, since `len` is not among the operators STP supports. Thus, Axe replaces `(len x)` with a cut variable. Here, the type of `(len x)` is given explicitly as an assumption, and so the STP query generated is:

```
NODE1 : BITVECTOR(1);
QUERY (
LET NODE2 = (0bin0 = NODE1) IN (
LET NODE3 = (0bin1 = NODE1) IN (
LET NODE4 = (NODE2 OR NODE3) IN (
(NODE4)))));
```

In general, there are three ways in which Axe can determine the type of a cut node. First, the node may have an obvious type. Most of Axe's operators carry explicit size parameters that give rise to such types. For example, the term `(bvxor '32 x y)` always represents a bit-vector of width 32.

Second, the type of a node may be known from the information provided in the clause, as for `(len x)` in the example above. Such information may be derived from explicit calls to type operators such as `unsigned-byte-p` or `booleanp`[10]. Furthermore, known equalities provide type information. If `x` and `y` are known to be equal and `x` has some type, `y`, then must have that type as well. More generally, since type information may be known for both `x` and `y` when their equality is encountered, Axe computes the intersection of their types[11] and stores the result as the type of both `x` and `y`. The type intersection operation can return the empty type (if `x` and `y` are known to have types that are incompatible), indicating that the assumptions of the goal contradict. (Such a goal is automatically considered proved, but a warning is printed.) The propagation of types through equalities can lead to further type information (e.g., through long chains of equalities), so Axe repeatedly processes the disjuncts of the clause, looking for improved type information, until no new information is discovered.

The third kind of type information arises from the chopping that Axe's bit-vector operators perform on their arguments. If a value is only used in such bit-vector contexts, it can be assigned a type corresponding to the largest number of significant bits used for any of its occurrences. In the example in Section 7.5.1, `x` and `y` have such "induced" types. Because they occur only as arguments to 32-bit XOR operators, any extra bits of `x` and `y` would be ignored. Furthermore, if either variable represented a non-bit-vector, it would be treated as 0 by `bvxor`. Thus, even without any explicit assumptions on their types, it is safe to assume that `x` and `y` are bit-vectors of width 32. STP's declaration that the resulting query is valid thus guarantees that the original formula is valid for all possible values of `x` and `y`.

[10] An array type would be represented by three separate facts, a `true-listp` claim, an `unsigned-byte-p-list` claim with a constant size, and a claim equating the length of the array with a constant.

[11] Types can be seen as forming a lattice.

7.6 Heuristic Cuts

Proofs regarding cryptographic code often include much irrelevant structure. For example, consider the problem of proving the equivalence of two implementations of an algorithm that operates as a sequence of "rounds." As described in more detail in Section 8.3, such a proof involves repeatedly merging equivalent subterms after proving them equivalent. Assume that at some point in the proof, the first n rounds of the two implementations have already been proved equivalent and merged. The next step would be to prove that the two implementations produce equivalent results for round $n + 1$. In such a situation, the term representing the result of the first n rounds would support both of the terms representing round $n + 1$. However, the computation of the first n rounds, already proved equivalent, is unlikely to be germane to the proof about round $n + 1$. Worse, this irrelevant term is likely to be large, and STP seems to bog down on queries that include large, complicated, irrelevant terms. Thus, Axe can heuristically cut out such subterms, even when they could actually be represented in STP's language. As is the case for operators that cannot be represented in STP, the cut nodes are replaced with fresh variables, and such a cut goal is more general than its corresponding uncut goal.

When the Axe Prover calls STP, several heuristic cuts may be tried in succession to attempt to prove the goal. First a proof with no heuristic cutting is attempted. If the uncut goal is proved by STP, the Axe Prover succeeds. If STP reports that the uncut proof is invalid, the Prover fails. (Since nothing was cut out, no more structure can be sent to STP.) The third possibility is that STP times out on the uncut goal. In such a case, the Prover attempts to find a good "depth" at which to place a cut. (The depth of a node is the length of the shortest path in the DAG to the node from the top node of any disjunct in the clause.) The Prover performs a binary search over all the possible depths. If at any depth, STP reports that the goal is valid, then the entire proof succeeds. Otherwise, if STP times out, a shallower depth is tried. The rationale is that too much structure was sent to STP but sending a smaller goal might succeed[12]. If STP reports that the goal at the current depth is invalid, a deeper depth is attempted. This is because any shallower cut would lead to a more general goal, which would also be invalid. However, the invalidity at the current depth may actually represent a false negative; in such a case, it makes sense to try sending more structure to STP.

The process of finding a cut depth requires achieving a delicate balance; the deeper the cut, the more likely the goal is to be valid, but the more likely STP is to timeout. Axe's use of binary search is an attempt to quickly find a depth at which the goal is valid but STP does not time out. If the attempt fails, then STP is considered to have timed out on the given case and the entire proof is abandoned (because every case must be proved).

7.7 Splitting Into Cases

When the Axe Prover cannot prove the goal as a single case, it splits into two cases, based on a subterm of the goal that is used in a boolean context (e.g., the test of an `if` or an argument to `boolor`). When there are several such candidates for splitting, the Prover chooses the one that corresponds to the smallest term (with sizes computed as if the terms were trees). To avoid loops, the prover avoids splitting on a node that is already a disjunct in the clause or the negation of a disjunct.

Splitting on `x` reduces the problem of proving the clause `C` to the equivalent problem of proving the two clauses `(or x C)` and `(or (not x) C)`. Each of the resulting cases contains one more piece of information than the original clause, and this can

[12] The implicit assumption here is that once a proof times out, sending more structure will also cause a timeout. This may not be true in all cases, but this whole approach is just a heuristic.

allow rewriting to make progress (for example, if the split fact expresses that an arithmetic computation rolls over).

Splitting in this way can cause a case explosion. Axe's approach is to only split when the other proof methods fail and to abandon the entire proof if any of the cases fails to prove. If STP times out when attempting to prove a case, that is considered failure and causes the entire goal to be abandoned.

The Prover takes as an argument the timeout to use for STP (a number of seconds, defaulting to 3). As described in Chapter 8, one of Axe's strategies for proving a miter after an initial failure is to increase the timeout and try again; this may help calls to the Axe Prover to succeed.

Chapter 8

The Axe Equivalence Checker

The Axe Equivalence Checker can prove the equivalence of two DAGs, given a set of assumptions about their inputs. If the checker reports success, it guarantees that the two DAGs have equal output values for all input values that satisfy the assumptions. The equivalence checker operates by forming a single DAG (called a "miter") representing the equality of the two input DAGs. The goal of equivalence checking is then to prove that this miter always evaluates to true (for any input that satisfies the assumptions). The rest of this chapter describes the basic equivalence checking process. If loop functions are present in the miter, extra steps are performed, as discussed in Chapters 10 through 13.

Note that, as described in Sections 9.3 and 10.4.1, symbolic execution and "opener rules" can often can be used to get rid of loops by completely unrolling them. In such cases, the basic equivalence checking process described in this chapter can be applied to the resulting unrolled computations. Such an approach can verify many implementations of block ciphers as well as cryptographic hash functions and stream ciphers applied to fixed-length messages.

Equivalence checking begins by rewriting the miter DAG. If rewriting fails to transform the miter into true, random test cases are used to discover probable equalities involving internal nodes. These internal equalities are used to break down the equivalence proof into a sequence of smaller proofs; the equivalence checker sweeps up the DAG, attempting to prove the probable equalities and, when successful, using the proven facts to transform the DAG. If the sweeping process fails to reduce the miter to true, the equivalence checker attempts to split the miter into two cases and then attempts to perform equivalence checking on each case, recursively. This chapter describes the equivalence checking process in more detail.

8.1 Mitering

A central operation in Axe (and many hardware equivalence checking tools) is "mitering," the formation of a "miter DAG" which represents the equality of two computations being compared[1]. The Axe Equivalence Checker takes as input two DAGs representing the computations to be proved equivalent. Corresponding input variables in the DAGs must be given the same names. (When comparing two implementations of a block cipher, each implementation might be expressed in terms of the variables `plaintext` and `key`.)[2]. Axe forms a miter DAG by merging the individual DAGs and adding a new top node for the equality of the two outputs[3].

The result of mitering is a DAG whose input variables are the inputs of the computations and whose output is a boolean value which will be true for any input

[1] The standard term is "miter circuit," but Axe usually deals with programs, not circuits.

[2] Extra but irrelevant input variables may appear in the DAGs, for example, to represent the initial contents of output buffers which will be overwritten without ever being read. The guarantee provided by the Equivalence Checker holds over all possible values of such variables that satisfy the assumptions.

[3] What if each computation has more than one output? Standard equivalence checking tools form miters by conjoining the equalities of each pair of corresponding outputs. In Axe, each function must have a single output, but that output can be a composite object (a tuple, list, or array). Such composite values can be equated directly; Axe will eventually rewrite such an equality into the conjunction of the equalities of the individual components.

which causes the computations to produce identical results. The goal of equivalence checking is to prove that the assumptions imply that this output is always true.

Because each subterm is represented only once in a DAG, the merging done when making a miter can lead to sharing of structure between the two implementations. This sharing is helpful when performing the equivalence check; any shared parts of the implementations have essentially already been proved equivalent. Indeed, most of the work done by the equivalence checker can be seen as increasing the sharing between the two "sides" of the miter, until they are completely merged, making their equality trivially true.

The amount of sharing in a newly-created miter can be significant and depends on how well the two computations have been normalized separately. In the extreme case of forming a miter for two isomorphic DAGs, the DAGs would be merged completely, and their two top nodes would be represented by the same node in the miter. The miter equality would then be of the form `(equal x x)`, which is clearly true. But note that any difference at all between the two DAGs will cause all nodes supported by the differing nodes to fail to merge, even if the structures above the point of difference are identical. In practice, there is usually some difference near the bottom of the DAGs which prevents much merging from being done when the miter is built.

Since the process of proving a miter simply entails proving that a large term is true, Axe sometimes applies equivalence checking to "miters" that do not actually result from equating two equivalent computations. For example, when proving that a loop function's body preserves its invariants, Axe forms the appropriate claim and attempts to prove it as if it were a miter. (It can be seen as a miter in which one side is the entire term to be proved and the other side is the constant "true.") The miter proving process can thus be seen as a general-purpose proof technique, which can be usefully applied to prove any claim for which breaking down the proof using internal equivalences might be helpful (or which might benefit from the techniques used to handle loop functions in miters, as described in later chapters).

A miter can be tested using an assignment of concrete values to its input variables. (Any such assignment should satisfy the miter's assumptions.) Using such an assignment, the nodes of the miter can be evaluated in a bottom-up fashion[4]. If the top node of the miter evaluates to false for any test case, that test case is a counterexample for the miter. If the miter represents the equality of two computations, the counterexample is a concrete input on which the two computations produce different results. In practice, there are almost always too many possible inputs to exhaustively test a miter. For example, the AES block cipher operates on a 128-bit plaintext and a key of at least 128 bits, giving at least 2^{256} possible inputs (more for longer keys). For cryptographic hash functions, there are even more possible inputs, because the length of the message is not fixed; Axe proofs about the SHA-1 and MD5 hash functions deal with messages of any length up to $2^{31} - 1$ bytes. Thus, testing is insufficient to prove equivalence of practical examples, so a proof must be done. The next sections describe how a miter is proved.

8.2 Rewriting the Miter

After forming a miter, Axe typically simplifies it via rewriting, using the Axe Rewriter described in Chapter 5. Even if the two DAGs have been simplified separately, it is often beneficial to simplify the miter immediately after creating it, for two reasons. First, some transformations may depend on node numbering, and that numbering is likely to differ between the two input DAGs. (When representing a term as a DAG there are usually many ways to assign node numbers to the individual subterms.) For example, the DAGs may contain equivalent nests of calls of some associative and commutative function, but the leaves in each nest may be sorted in different orders, as

[4] As explained in section 8.3.1, Axe's evaluation is actually more sophisticated than the simple bottom-up algorithm.

dictated by the node numbering in their respective DAGs. Simplifying after creating the miter will cause both nests to be rewritten using the same node numbering (that of the miter DAG). Thus, both nests may rewrite to the same thing. Simplifying the newly created miter thus increases sharing and makes the two computations more similar. Furthermore, the effect can ripple up; nodes above the newly merged subterms may be able to be merged as well.

The second benefit of rewriting the miter is that doing so can simplify its top node, which is usually an equality. If the node expresses the equality of composite objects (tuples, sequences, or arrays), rewriting can transform the equality into a conjunction of equalities of smaller terms, each of which can then be proved separately. Rewriting the equality can also allow cancellation rules to be applied, perhaps discarding irrelevant portions of the miter.

When Axe simplifies a miter, it uses its large library of rewrite rules, including the rules to handle common coding idioms (described in Chapter 6). Typically, Axe simplifies the miter first using word-level operators, then performs bit-blasting, then simplifies the bit-blasted miter again. There are several advantages of rewriting at the word level before bit-blasting. First, the DAGs involved are smaller. More importantly, some properties are much more apparent at the word level than they are after bit-blasting. For example, at the word level it is convenient to transform `bvplus` nests using the associative and commutative properties, but once the `bvplus` operations are bit-blasted (which essentially turns each one into a ripple-carry adder), such transformations cannot easily be perfomed.

For several block cipher implementations, rewriting the miter as described in this section is sufficient to prove it. That is, rewriting causes the implementations to be normalized to the same thing. They thus merge completely and the miter equality rewrites to true.

However, in many cases rewriting alone is not sufficient to prove the miter. This is perhaps not surprising, since Axe's library of rewrite rules is not guaranteed to produce normal forms. In actual proofs involving cryptographic code, rewriting alone tends to fail to prove equivalence when there are large differences between the two implementations being compared. This occurs especially when lookup tables are used to replace sequences of logical operations. Even if it doesn't completely prove the miter, rewriting may nevertheless increase sharing between the two implementations, thus helping with next step of the equivalence check: sweeping and merging.

8.3 Sweeping and Merging

The success of the Axe Equivalence Checker on large examples often depends critically on the presence of internal correspondences between the two implementations being compared. That is, some of the intermediate values computed in one computation should correspond to values computed in the other computation. These correspondences will manifest themselves as pairs of nodes – one on each "side" of the miter – which are always equal under any input to the miter[5]. Axe evaluates random test cases to identify such pairs of nodes. Of course, this testing is not exhaustive and so cannot establish the equality of the nodes; thus we say they are only "probably equal." During the testing process Axe also identifies nodes which are "probably constant." Such a node evaluates to the same value on every test case.

If during the testing process Axe ever encounters a test that makes the top node of the miter false (but satisfies the input assumptions), that test case represents a counterexample to the claim of equivalence, and the equivalence checking process fails immediately.

After determining the "probable" facts, the Axe Equivalence Checker sweeps up the DAG in a bottom-up fashion. When it encounters a node that is probably equal

[5] Actually, miter nodes may not be "used" on all test cases, due to `if` operators above them in the DAG. For nodes to be considered equivalent, they must be unused on the same set of test cases.

to something else (another node or a constant), it attempts to prove that fact. If it succeeds, a merge is performed. To merge two nodes that have been proved equivalent, Axe changes all mentions of one node to instead mention the other node, thus making the two sides of the miter more similar. To merge a node with a constant, Axe simply replaces all mentions of the node by the constant. This makes the miter simpler, because a constant is considered simpler than any other expression. In both cases, after a merge, the node that was replaced is no longer mentioned in the DAG. It (and perhaps some of its supporters) have been made irrelevant. The sweeping process continues up the DAG. Eventually, if all goes well, the top nodes of the two implementations will be merged. (They will always be considered "probably equal" unless a counterexample is found.) Then the top equality (which will always be considered "probably true") can be trivially proved. The sweeping process can be visualized as the action of a zipper; at a given point, equivalent nodes below some frontier in the DAG have been merged together, and the nodes above the frontier are yet to be merged.

The sweeping and merging process works well in the verification of cryptographic code, because there are usually internal correspondences between two implementations of the same algorithm. For example, a block cipher is usually structured as a sequence of rounds, with the values computed in each round serving as input to the next round. Different implementations may differ significantly in how they compute the rounds (e.g., in the order in which operations are performed, or by replacing sequences of logical operations by table lookups). However, the values passed from one round to the next almost always correspond between two implementations of the same cipher. Axe exploits these correspondences to break down the equivalence proof for a cipher into a series of smaller proofs. While the full proof for a cipher is usually intractable without sweeping and merging, each of the small proofs (which amount to proving the equivalence of one round or partial round) is usually tractable (if the proofs are performed with the heuristic cutting of STP queries described in Section 7.6).

It remains to describe the details of the sweeping and merging process, including how tests are generated and evaluated, how probable facts are detected, and how the proofs supporting the merges are actually performed.

8.3.1 Evaluating Random Tests

In order to detect probable facts about DAG nodes, Axe repeatedly assigns values to the nodes of the miter. Each valuation is computed using randomly generated values for the miter's inputs, and Axe takes care to include only values that can actually occur on real, valid inputs to the programs being compared. This section describes how such inputs are generated and used to assign values to DAG nodes.

Top Level and Nested Miters

Miters in Axe can be divided into two groups. "Top-level" miters, the focus of this chapter, are used to reason about entire programs (e.g., to prove two implementations of the same cryptographic algorithm equivalent). The variables of top-level miters are simply the input variables to the programs being compared. Test cases for such miters are thus test cases for the entire programs, and the assumptions used when manipulating those miters correspond to the preconditions of the programs.

Axe also supports "nested miters," which are miters used to prove properties of loop bodies (such as the fact that a loop body preserves an invariant, or that the bodies of two loops preserve some connection between the loop variables). The basic equivalence checking approach described in this chapter can be applied to nested miters, except that generating test cases for them is more complicated. Section 11.5.3 discusses how test cases for such miters are generated from the execution traces of loop functions. The key point is that the test cases ultimately correspond to real behaviors of the top level programs.

Generating Random Inputs

Axe uses user-supplied type annotations to generate test cases for top-level miters. The goal is to generate test cases that satisfy the assumptions about the miter, that are not too large, and that reveal interesting behaviors.

It is often clear how to generate random values that satisfy the assumptions of a miter. For example, the assumption `(unsigned-byte-p '8 x)` can be satisfied by generating random 8-bit values for `x`. However, sometimes satisfying the assumptions is more difficult. In particular, it is common in cryptographic code for the type of one input to depend on the type of another input. Usually, this is because the algorithm takes a variable-length message and a buffer into which the output will be written, where the length of the output buffer depends on the length of the input. For example, the RC4 cipher's output buffer must have the same length as the message[6]. In such cases, if both the input and output lengths were independently randomly generated, the output buffer would rarely have the correct length. To better support such cases, Axe allows the types of random values to depend on previously generated values.

Axe takes as input an association list (or "alist") mapping the variables of the top-level miter to types. Random values are generated for variables in the order in which they appear in this association list, and so the type for a variable V can depend on the value given to any variable that precedes V in the alist. Other ways to handle this issue might include solving the assumptions (which may be hard in general) or simply generating test cases until all of the assumptions are satisfied (but dependencies might make that process take too long).

[6] A more complicated example is the repeated application of a block cipher according to some "mode of operation," for which the output buffer must be sized to contain extra padding bytes. The buffer's length is thus a somewhat complicated function of the input length.

There are also cases in which a miter's type annotations need to differ from its assumptions. Some cryptographic programs allow inputs that are very large, and proofs about such programs should cover those cases. For example, the Axe proof about the RC4 stream cipher covers input messages up to $2^{31}-1$ bytes long. Naturally, the assumptions of the miter allow for such very long messages. However, it would be undesirable to generate such long messages as test cases, since the tests would take a very long time to run. Thus, the type annotation for a variable may be more restrictive than indicated by the assumptions. Usually, small test cases (ones that perform just a few loop iterations) are sufficient to reveal the behaviors of interest to Axe[7].

Supplying an annotation for the type of each variable is not too onerous because it is required only for the top level miter, and because the annotations are often closely related to the miter's assumptions. (Still, the process could perhaps be automated.)

Here is the type annotation for the RC4 stream cipher:

```
((input . (:list (:bv 8) (:range 0 512)))
 (output . (:list (:bv 8) (:eval (len input))))
 (key . (:list (:bv 8) (:range 1 256))))
```

It declares `input` to be a list of random 8-bit bytes whose length is a random number between 0 and 512. Lists longer than 512 bytes are thus disallowed for testing purposes.. The `output` is also a list of random 8-bit bytes, and its length depends on `input`. The annotation `(:eval (len input))` indicates that Axe should simply evaluate the application of `len` to the random value generated for `input` and then use the result as the length of `output`. This requires `input` to precede `output` in the

[7] A notable phenomenon occurs with algorithms, such as SHA-1, in which the message length is actually included as part of the padding data (and then processed as data by the main algorithm). Because only small test cases are used, Axe will incorrectly believe that the upper bits of the message length are always 0. This causes Axe to waste time trying and failing to prove those spurious claims. However, it does not prevent the entire proof from completing.

alist. The bytes in `output` are irrelevant to the computation (because they will be overwritten with the ciphertext), but using random values (as opposed to, say, zeros) prevents Axe from incorrectly trying to prove any facts about those bytes. Finally, the `key` is declared to be a list of random bytes whose length is a randomly selected integer between 1 and 256. (This covers all of the "interesting" RC4 key lengths.)

Axe's type annotations also allow biasing of the generated values toward interesting cases. This can help exercise behaviors that are relatively rare. For example, the following annotation for an input message:

```
(input . (:list (:bv 8) (:choice (:range 0 16) (:range 0 512))))
```

specifies that the length of `input` should half of the time be a random integer in the range [0,16] and half the time be a random integer in the larger range [0,512]. This specification could be used to increase the number of short messages generated if such messages are particularly likely to reveal interesting behaviors.

The following table describes all of the possible type annotations in Axe:

Table 8.1: Grammar for Random Test Cases

Type	Value
`(:bv <width>)`	a random bit-vector of width `<width>`
`(:list <element-type> <length-type>)`	a list (or array) whose length is given by `<length-type>` and whose elements are of type `<element-type>`
`(:range <lower-bound> <upper-bound>)`	a random integer in the inclusive interval `[<lower-bound>, <upper-bound>]`
`(quote <val>)`	the unquoted value, `<val>`
`<symbol>`	the value of variable `<symbol>`, which must have already been given a value
`(:eval <expression>)`	the result of evaluating the expression `<expression>` with all the variables bound to their already-generated values
`(:choice <type1> <type2>)`	a value whose type is randomly chosen to be of `<type1>` (half the time) or `<type2>` (half the time)
`(:element <val1> <val2> ... <valn>)`	a randomly chosen element of the set {`<val1>, <val2>, ..., <valn>`}

Assigning Values to Internal Nodes

Given values for the input variables of a DAG, Axe can compute values for its internal nodes. The values are memoized, so no node is evaluated more than once per test case.

In order to evaluate a normal function call node, Axe first evaluates all of its children, but special handling is applied for if-then-else nodes. Axe evaluates the condition of the if-then-else and then evaluates only the appropriate branch (then-branch or else-branch, depending on whether the condition is non-`nil`). This means that not all nodes will be "used" (i.e., have their values computed) for each test case. In addition to saving time, refraining from evaluating certain nodes can prevent the generation of meaningless values (e.g., the value of a division operation on a branch in which the divisor is 0)[8], which might unduly invalidate "probable" facts.

During the testing process, all of the tests for a miter must make the top node of the miter true. If any test makes it false, that test is a counterexample for the miter. If such a counterexample is found, the proof of the miter fails. Furthermore, if the miter is a top-level miter, an error is thrown, since, unlike nested miters (which are allowed to fail), the proof of a top-level miter should never fail.

8.3.2 Computing Probable Facts

By examining the values of DAG nodes on many test cases, Axe discovers "probable facts." These are facts that hold over every observed test case, and they come in two types: claims that nodes are constant, and claims that pairs of nodes are equal. If a fact holds on every test case run by Axe, Axe considers the fact to be probably true. Of course, this does not constitute a proof. There may be inputs not among those tested that would falsify the fact. So all such facts must be proved. If a probable

[8] Some functions which model loops in Java code may not even terminate if given "bad" inputs (for which the corresponding Java code would loop forever).

fact can actually be formally proved, it can help Axe perform equivalence checking, because the nodes can be merged. The rest of section describes the efficient detection of "probable facts."

Axe conjectures that a node is "probably constant" when it has the same value on every test case for which it is used (and if it is used for at least one test case). To efficiently compute probable constants, Axe uses two data structures, a "never-used" list and an association list pairing nodes with the constant values that they seem equal to. Axe begins by running a single test case. The unused nodes become the initial elements of the never-used list, and each used node is paired in the alist with its value on the first test case. Additional test cases are used to update these data structures as follows. If a node on the never-used list is used on a test case, it is moved to the alist and paired with its value on that test case. If a node is already paired with some value in the alist and is used on a test case but has a different value, its pair is removed from the alist. (The node has been observed to have two different values and so is not considered probably constant.) After all of the test cases have been run, any pairs remaining in the alist correspond to probable constants. Note that the total number of nodes under consideration (in the never-used list and the alist) decreases over time. So later test cases are usually processed faster than earlier test cases. Also, the value of a node on a test for which it is not used does not invalidate probable constants.

Axe also uses test cases to partition the nodes of the DAG into "probably equal" sets. Two nodes are placed into the same set if they are used on the same test cases and have the same value on every test case for which they are used. Axe computes this partition efficiently as follows. First, a single test case is run, giving values to the nodes (including the special value `:unused` for unused nodes). Nodes are paired with their values, the pairs are sorted by these values, and nodes with the same value are grouped together. The resulting groups are the initial probably-equal sets. Any singleton sets are removed. Each test case is then used to try to split each probably equal set. Whenever the nodes in a set do not all have the same value on the test case (again, `:unused` is considered a value), the set is split into one or more new sets (depending on the number of different values assumed by the nodes on the current test case). Again, any singleton sets are removed. Finally the set of nodes never used on any test case (if present) is dropped. The remaining sets are the probably-equal node sets. As the algorithm runs, the partition gets increasingly fine, and the number of nodes still under consideration decreases. Thus, again, the algorithm speeds up as it goes. The critical thing about the algorithm is that it does not exhibit any quadratic behavior (in n, the number of nodes in the DAG), even though the number of possible pairs of nodes is $O(n^2)$.

Axe's procedure to compute probable equalities requires probably-equal nodes to be used on exactly the same test cases, because it is not clear how to relax that restriction and still efficiently compute the sets[9]. This policy seems to account for the slow speed of the full Axe proofs about the SHA-1 and MD5 hash functions. Those proofs take days to run because the miters must be split into cases, and the case splitting seems to be necessary because if-then-else nodes on one side of each miter cause nodes to sometimes be unused. The sometimes-unused nodes are not considered probably-equal to nodes on the other side of the miter.

As it executes test cases, Axe keeps track of when the last "interesting" test case occurred (i.e., the last test case that invalidated any probable fact). If the number of test cases executed increases by a factor of 10 without any change to the probable facts, and if at least 100 test cases have been executed, Axe concludes that executing further tests would likely be uninteresting and so ends the testing process immediately.

[9] Consider splitting a set on a test case for which some nodes have the value 0, others have the value 1, and still others are unused. Into which of the resulting sets should the unused nodes go?

8.3.3 Proving a Merge

Axe employs a variety of strategies for proving the equality of nodes that are considered “probably equal.” First, if the nodes already contain identical expressions (because their arguments have already been proved equal and merged), they are considered to be “structurally identical,” and the proof succeeds immediately. Otherwise, some work must be done to perform the proof. The structure of the proof depends on the operators that occur in the miter. Either the miter is “pure,” meaning it (and its assumptions) only contain operations which can be translated into the language of STP, or it is impure. Pure miters result from completely unrolling cryptographic programs and are usually large but contain relatively simple operators. Impure miters usually arise in proofs done without complete unrolling. They usually have fewer nodes but require deeper analysis because they contain loop functions.

When proving the equality of two nodes in a pure miter, the emphasis is on speed because there are likely to be many node pairs to handle. To do the proof, Axe calls STP. The process is similar to when the Axe Prover calls STP (described in Section 7.5), with two differences. First, no complicated procedure is needed to infer the types of nodes, because in a pure miter all nodes have obvious types. Second, the cutting heuristic is slightly different; Axe first uses a very aggressive cut which cuts out any node that is a shared supporter of the nodes being proved equal. (Intuitively, this means cutting out the part of the DAG that has already been merged.) If this fails, Axe uses the binary search heuristic (described in Section 7.6) to attempt to find a suitable cut depth.

When dealing with an impure miter, more work is expended to attempt to prove each merge. This is appropriate because such miters usually have fewer node pairs to merge but contain loop functions that require sophisticated analysis and are not representable in the language of STP. We defer the detailed discussion of this case until section 10.5.

8.3.4 Merging the Nodes

Once two nodes have been proved equivalent, Axe "merges" the nodes. That is, it chooses one of the nodes to eliminate and changes all references to that node to instead point to the other node. In order to preserve the node ordering property of the DAG, Axe always eliminates the node with the higher node number. Its parents will then be changed to point instead to the node with the lower node number, preserving the property that child nodes have smaller numbers than their parents.

Once a node has been eliminated, it (along with any of its supporters that do not support other nodes still in the DAG) has essentially been removed from the DAG, and the two "sides" of the miter have become more equal.

In order to be fast, merging of two nodes is done by changing the underlying DAG. This departs from Axe's usual behavior of building new nodes instead of changing nodes in place. It may even violate the property that two DAG nodes never contain the same expression. Consider two nodes `(foo x)` and `(foo y)`[10]. If `x` and `y` are merged (with `x` chosen as the node to be eliminated), the call to `(foo x)` will become identical to `(foo y)`. The calls to `foo` could then be merged, perhaps enabling future merges if Axe propagated such information upward in the DAG. However, Axe instead relies on the quick check for structural identity (mentioned in the previous section) to handle such cases when it comes time to merge the pairs in question.

An even more aggressive approach would be to rewrite the DAG after each merge. Axe avoids this in order to keep the merging process very fast. (Proofs about unrolled ciphers involve hundreds or thousands of merges.)

[10] represented here as terms only for clarity, of course.

8.4 Splitting the Miter

If the equivalence checking process described so far is not able to prove the miter (because nodes failed to be proved equal and merged), Axe can split the miter into two cases. It does so based on a node used in a boolean context (e.g., the condition of an if-then-else operator). The split node and its negation are each added as an assumption in one of the cases, and both cases are processed by recursive calls to the equivalence checker. (Such case splits provide a clear way to parallelize Axe proofs. Since the cases are independent, they could be processed on different machines or different processor cores. Axe does not currently exploit this opportunity, but doing so would greatly speed up the expensive Axe proofs about cryptographic hash functions.)

To choose the split node, Axe picks the candidate splitter that corresponds to the smallest term (measuring the sizes of the trees represented by the nodes), with one additional restriction. It never splits on a node that is either always true on every test case or always false on every test case. Doing so would lead to a split for which one of the cases had no test cases that satisfied its assumptions, and test cases are vital to the equivalence checking process.

After adding a split fact (or its negation) to the assumptions of a miter, Axe rewrites all of the assumptions in an attempt to strengthen them using the new fact. This is done by passing the rewrite objective `nil` (meaning to strengthen) to the Axe Rewriter (as discussed in Section 5.13). The process continues until no assumptions can be further strengthened. This rewriting process does not change the logical strength of the conjunction of the set of assumptions, but it may strengthen individual assumptions and make them better suited for use in rewriting the miter.

Splitting a miter into cases is helpful in several scenarios, because it can eliminate if-then-else operations from the DAG. As described in Section 8.3.2, such operators can interfere with the identification of probably-equal nodes if they appear on only one side of the miter. Also, if-then-else operators are sometimes interposed between operators whose composition could otherwise be simplified. Consider the term:

```
(car (if test (cons a b) (cons c d)))
```

If a split on `test` occurs, the cases would be

```
(car (cons a b))
```

and

```
(car (cons c d)),
```

each of which could be simplified. (The same effect could be accomplished by "if lifting" but, like case splitting, that operation can be expensive.)

Chapter 9

Reasoning About Java

The core of Axe operates on computations represented as DAGs. Thus, to verify a program in a real-world programming language, such as Java, one must extract a DAG representing the computation performed by the program. The DAG should express the program's output in terms of the program's inputs, using only Axe's built-in operators, and perhaps loop functions. Capturing the input-output behavior of a computation in a DAG allows the core of Axe to avoid dealing directly with the semantics of Java or Java bytecode. Since its core analysis is language-independent, Axe could also be applied to DAGs generated from programs in another language[1]; one would simply need a way of extracting DAGs representing the input-output behavior of programs in that language. This chapter describes how Axe extracts such DAGs from Java programs.

Axe handles Java programs in two main ways: by complete symbolic execution (for programs whose loop iteration counts are all fixed, allowing the loops to be completely unrolled), and by decompilation (which can handle non-unrollable loops and produces DAGs that contain calls to loop functions). Both of these techniques rely on the Axe Rewriter and a formal model of the Java Virtual Machine. (Some of

[1] It already does so, if you count the formal specifications written in ACL2.

the machinery described in this chapter may be useful to others who wish to reason about Java code, even if they do not with to use Axe.)

9.1 Java Bytecode and the JVM

In order to run a Java program, one first compiles it to Java "bytecode," an assembly-like language designed to run on the Java Virtual Machine (JVM). Because the JVM has been implemented on many different platforms, the resulting bytecode is portable. The semantics of Java bytecode are precisely defined in the book *The Java Virtual Machine Specification* [40], which explains bytecode in sufficient detail that one can use it to create an implementation of the Java Virtual Machine.

In order to reason formally about Java code, it is necessary to formally specify the semantics of the Java language. Others have sidestepped this issue by instead dealing with the compiled bytecode. For example, J Moore's research group has produced a formal, executable model of the JVM, called M5 [47], and they use it reason about Java programs. Axe takes a similar approach. In fact, its model is based on M5, as described below. Reasoning at the bytecode level has the additional advantage that the code that is checked is the code that actually runs on the JVM. Errors in the compilation process would thus be caught, whereas reasoning at the Java level would not catch them. Also, bytecode verification can be performed even when the source code is not available.

The bytecode representations of Java classes are stored in class files (files with extension `.class`). Each class file contains a good deal of information, including the code of the class's subroutines. (Java subroutines are called "methods.") The information is stored as a sequence of bytes, and Axe contains a parser which transforms a class file into a structured Lisp object representing the class. The first steps in using Axe to reason about Java code thus consist of compiling the code and parsing the resulting class files. (The Axe class file parser could be used as a stand-alone tool by others who wish to obtain structured versions of Java class files.)

9.2 M5E: The JVM Model

Axe uses a formal model of the Java Virtual Machine called M5E, which is based on the M5 model mentioned above. In some sense, M5 and M5E are just implementations of the JVM, albeit ones that can be reasoned about formally.

M5E represents the execution state of the Java Virtual Machine as a Lisp object, namely a tuple whose components include a heap, a class table (containing the code and attributes of all classes and methods), and a call stack that keeps track of currently-pending method invocations[2]. Each frame of the call stack corresponds to the in-progress execution of some method. The frame is pushed onto the stack when the method is called and popped off when the method returns. Each frame contains data required for the execution of the method, including local variables and an operand stack used for expression evaluation[3].

M5E includes definitions reflecting how the execution of each Java bytecode instruction[4] affects the JVM state. For example, the integer addition operation, `IADD`, is modeled as follows:

```
(defun execute-IADD (th s)
  (modify th s
          :pc (+ 1 (pc (top-frame th s)))
          :stack (push (acl2::bvplus 32
                                     (top (pop (stack (top-frame th s))))
                                     (top (stack (top-frame th s))))
```

[2] Actually, there is one call stack for each thread, but the programs verified by Axe are not multithreaded.

[3] The JVM is a “stack machine” and uses the operand stack, instead of registers, for holding intermediate results.

[4] Some instructions are not yet supported, including the ones that deal with floating point numbers.

```
                    (pop (pop (stack (top-frame th s)))))))
```

The function `execute-IADD` modifies the data structures of thread `th` in the given JVM state `s`. In particular, it pops two operands off of the operand stack, adds them, and pushes the sum back onto the stack. It then increments the program counter (`:pc`) by 1 (the length of the `IADD` instruction).

M5E contains several dozen such functions, each for a different Java bytecode instruction (many of which are more complicated than `IADD`). M5E also contains a function, `do-inst`, which fetches the next instruction to be run (using the program counter) and then calls the appropriate execution function. Finally, M5E contains several functions that repeatedly call `do-inst`, including `run-until-return`, which repeatedly steps the state until the call stack height is less than it was to start with[5]. That is, `run-until-return` runs the current method until it pops off its stack frame and returns. (If the current method calls other methods, the call stack height will grow temporarily but will eventually decrease, assuming the main method eventually returns.)

M5E is a formalization of the informal descriptions in *The Java Virtual Machine Specification.* It thus constitutes a formal, operational semantics for (most of) the Java bytecode language. As mentioned, M5E is quite similar to the M5 model. However, M5E contains many technical improvements designed to improve the clarity of the model, increase the ease of reasoning about it, and increase the speed of symbolic executions. One change is worth noting here; M5E uses Axe's operators to model arithmetic and logical operations. Thus, the DAGs extracted from Java programs (as described in the next section) will use Axe's built-in operators.

[5] How can such a function terminate if the Java code contains an infinite loop? Actually, `run-until-return` is defined in terms of a partial function, `run-until-return-from-stack-height`, which can be introduced soundly using ACL2's `defpun` feature [41].

9.3 Symbolic Execution Via Rewriting

For a program whose loop iteration counts are all fixed, Axe can perform symbolic execution (sometimes called "symbolic simulation") to extract a DAG representing the program's output. The process uses the Axe Rewriter and the M5E model. The Rewriter repeatedly rewrites a term representing the execution of M5E on a concrete program, with each rewriting step corresponding to the execution of one instruction. Axe's approach borrows heavily from the approach of Moore and others, which uses M5 and the ACL2 rewriter. A crucial difference is that Axe represents terms as DAGs and can thus compactly express the results of unrolled cryptographic computations. Such expressions are vastly too large for ACL2's tree representation.

The starting point of a symbolic execution in Axe is a term representing the state of the JVM just after the invocation of the method of interest. In such a state, the program being executed is known, and the program counter is known to be 0. However, the data to be operated on (represented as local variables and values in the heap) is arbitrary (represented by symbolic variables, hence the term "symbolic execution"). The operation of running the method until it returns is modeled by a call of `run-until-return` applied to the JVM state. In essence, symbolic execution involves repeatedly rewriting the call to `run-until-return`, with the state term evolving as a sequence of bytecode operations (the program for the method) is performed on it.

The key fact used during rewriting can be expressed as follows: If the method in question has not yet returned, then running the state until the method returns is equivalent to calling `do-inst` to execute the next instruction and then running the resulting state until the method returns. This fact is represented as a rewrite rule[6], and it is applied repeatedly. Each application of the rule introduces a call of `do-inst`, and, because the instruction being executed is always known, each call of `do-inst`

[6] The requirement that the method has not already returned makes it a conditional rewrite rule.

is expanded into a call of the appropriate execution function, such as `execute-IADD`. Simplification is aggressively performed on the intermediate states, and the net effect is to build up symbolic terms which represent the values of various components of the JVM state (local variables, values in the heap, etc.) in terms of the inputs to the program.

This approach works well for straight-line code, but conditional branches can cause problems. In particular, when a conditional branch is reached, it may not be possible to "resolve" the branch condition by rewriting it to either true or false. Different branches may be taken for different inputs, and symbolic execution represents the handling of arbitrary inputs. When a branch condition cannot be resolved, the symbolic execution will have to split. That is, the term representing the JVM state will become an if-then-else term whose branches are state terms with different program counters. Symbolic execution will then continue separately on each of the two branches. Of course, there may be more than one conditional branch in a program, and the resulting if-them-else nests may become unwieldy.

Symbolic execution works well for cryptographic programs (such as most block ciphers) that contain few non-resolvable conditional branches. In particular, when the loop iteration counts of a program are all fixed, the loop continuation tests (which determine whether each loop executes again or exits) can be resolved. For example, consider an implementation of AES encryption that performs 10 rounds, corresponding to values of 0 through 9 for the loop index, `i`. The loop continuation test would simply check whether `i` is less than 10. Before symbolic execution reaches the loop, `i` would be initialized to 0. The loop continuation test would then be resolved to true, because 0 is less than 10. Thus, execution would continue, executing the loop body, incrementing `i`, and eventually reaching the loop continuation test again. This time, `i` would be 1, which is still less than 10, so execution would continue. Eventually, after 10 iterations, `i` would be 10, and the continuation test would be resolved to false, because 10 is not less than 10. The loop would then exit. The result of the symbolic execution would be a large term representing the computation done by all 10 loop iterations (essentially, it would be the simplified composition of 10 copies of the loop body, with the appropriate value of `i` used for each). Importantly, the result would not include a loop (or a recursive loop function). Such a symbolic execution thus amounts to completely unrolling the loop.

When non-unrollable loops are present, the process just described will not work, because the loop continuation tests will not be resolvable. Consider a loop for which the number of iterations depends on the input. When symbolic execution reaches the loop continuation test, the test will not be resolvable, because for some inputs (e.g., an input with no more data to left process), the loop will exit, while for other inputs it will execute again. Applying Axe to such programs requires the bytecode decompiler described in the next section.

9.4 The JVM Bytecode Decompiler

Axe's bytecode decompiler allows the extraction of DAGs from programs with non-unrollable loops; the resulting DAGs will contain loop functions (as described in Section 4.11). The decompiler relies on symbolic execution of segments of code using the Axe Rewriter and the M5E model.

Essentially, the decompiler takes as input a collection of classes and an indication of which method is to be decompiled. It returns a DAG representing the complete execution of the method on arbitrary symbolic data. If loop functions appear in the decompiler's output, it also emits definitions for those loop functions (and for any nested loop functions they contain, and so on). This section explains the decompilation process at a very high level and elides many details. The main point is that Axe's decompiler provides an example of how to build a decompiler for a language from an operational semantics for the language and a general-purpose rewriter.

The decompiler begins by identifying all of the loops in the bytecode to be decompiled. In particular, it identifies loop "headers," which are instructions at which loops are entered. A central operation of the decompiler is to decompile a segment of code (such as the body of a method or loop) until a loop header is reached. If there are no loops in the segment, decompilation is essentially the same as the symbolic execution described in the previous section. However, when symbolic execution reaches a loop header, the execution stops, and the decompiler handles the loop (yielding a representation of how the loop changes the JVM state). Decompilation of surrounding segment then continues.

It remains to explain how Axe decompiles a loop when standing at the loop header. This process is mutually recursive with decompiling a code segment. In particular, to decompile a loop, Axe first decompiles the segment representing the loop body (including any nested loops). The result is a representation of the effect of the loop body on an arbitrary state. This will be an if-then-else nest containing at least two branches. There will be a branch for which control exits the loop and a branch for which control returns to the loop header. (There may be more than two branches, corresponding to additional code paths through the loop body.) The decompiler uses this expression to build the recursive loop function that models the loop. The function's parameters will be any state components (such as local variables or fields of heap objects) which are changed by any path through the loop body. The loop function's exit test can be derived from the structure of the if-then-else term; it will contain the conditions governing all branches for which control exits the loop. The body of the loop function (that is, the expressions used to calculate new values for each of its parameters) can be derived from those branches of the if-then-else nest that do not exit the loop. (If there are multiple code paths through the loop, the updated values of state components will be if-then-else expressions.)

Given the parameters, exit test, and update expressions, Axe builds a tail-recursive simple loop function to represent the loop. Recall that the loop was decompiled when control reached its header. The decompiler can now use the new loop function to represent the entire execution of the loop. In particular, executing the entire loop is the same as reading the necessary components out of the JVM state, then calling the loop function on them, and then writing the results back into the correct components of the state. Using the loop function to characterize the loop in this way allows the symbolic execution to jump past the entire loop and continue until the next loop header is reached.

The previous description omits several details. For example, symbolically executing a loop body on an arbitrary state is not always possible. For example, the decompiler may need to know the classes of certain heap objects in order to resolve method calls on those objects. Thus, the decompiler detects some simple invariants, which it uses during decompilation of the loop body. It then checks that the invariants were preserved by the loop body. Also, the decompilation process deals with exceptions in an unsound way; it simply discards any execution branches that throw exceptions. Thus, Axe's correctness proofs about Java programs only hold when exceptions are not thrown[7]. Also, Axe simply assumes that all loop functions terminate. Since a non-terminating loop function could technically render the ACL2 logic unsound, Axe's correctness results should be considered "partial correctness" results in the technical sense; Axe guarantees correctness if the programs terminate.

Axe's decompiler cannot handle programs that perform complicated heap manipulations (such dealing with linked lists). Each loop function produced by the decompiler has parameters that correspond to individual state components (bit-vectors or arrays), but decompiling a complicated heap-manipulating program would require

[7] Axe's handling of exceptions could be improved, or other techniques could be used to prove that programs never throw exceptions for valid inputs.

one of the function's parameters to be a heap, which would greatly complicate its subsequent analysis.

9.5 Driver Programs

A Java program that uses another Java class (such as those verified by Axe) to perform cryptography often must perform several steps, Usually, it must first allocate an object of the cryptographic class. (This causes the constructor to be executed and class initialization to occur.) Then it may call a method to initialize the cryptographic engine, perhaps passing in the encryption key to be used. Then the client program will call the method that performs the bulk of the computation. Finally, it may call a method that finalizes the computation. To represent the sequence of operations to be verified, the user of Axe writes a small "driver" method that performs all of these steps[8]. Axe is then applied to verify the input-output behavior of the driver, proving it equivalent to a formal specification (or to the driver of a second Java implementation). When symbolic execution or decompilation is performed, it is actually done on the driver method. (Of course, the process involves symbolically executing or decompiling all of the methods called by the driver.) The Axe proof about a Java program thus covers only the "usual" sequence of method calls performed by the driver; it does not cover other ways of interacting with the class.

[8] If you can write code that uses the cryptographic class, you can write the driver.

Chapter 10

Dealing with Loops

The functions that appear in miters may be divided into several classes, depending on how they are handled by Axe. First are the "built-in" functions, which deal with bit-vectors, arrays, booleans, and sequences. Axe reasons about such functions using rewrite rules and by calling the external tool STP.

Functions which are not built-ins are called "user functions." User functions appear in formal specifications and in the output of the JVM bytecode decompiler. The Axe approach is to expand all non-recursive user functions using their definitional axioms. It remains to discuss recursive user functions. Such functions typically represent loops in code (as described in Section 4.11) and so are called "loop functions." Axe includes two major approaches to reasoning about loop functions. The first is to completely unroll them, if possible, yielding large loop-free terms which are then processed by the Equivalence Checker. The second approach is to deal with loop functions using inductive methods, by finding and proving inductively-strong loop invariants. Loop invariants are discovered by examining traces. Often when reasoning about loop functions, transformations must be performed to either simplify the functions involved or to synchronize two loop functions being compared. This chapter gives an overview of Axe's process of reasoning about loop functions.

10.1 Inductive Proofs In Axe

Most of the Axe's reasoning about loop functions involves proof by induction. For example, induction is required to prove that a loop function satisfies an invariant, that transforming a loop function into a simpler function preserves its behavior, or that two loop functions implementing the same algorithm are equivalent. The ACL2 theorem prover provides excellent support for proof by induction[1], and when Axe does induction proofs it relies on the induction support of the underlying ACL2 system. Typically Axe proves the inductive step (and the base case) of the proof and then relies on ACL2 to put the pieces together. The inductive step (for example, showing that the loop bodies of two loops preserve some connection between the loop variables) can usually be proved without induction (unless there are nested loops) but may involve large terms that require the use of Axe's DAG representation and the Axe Equivalence Checker. The inductive step usually deals with a single loop iteration, and ACL2 is invoked to extend the proof to cover an arbitrary number of loop iterations (by induction).

Rather than mathematical induction (which is used to prove claims for all natural numbers), ACL2 proofs about recursive functions use "structural induction," in which the proof obligations are generated from the structure of the recursive functions being analyzed. In an ACL2 proof about a simple loop function (which is always tail-recursive), the inductive step of the proof is to show that the updates performed on the arguments before passing them on to the recursive call preserve the relevant property. This corresponds to showing that one iteration of the body of the corresponding Java loop preserves its loop invariants.

When more than one recursive function appears in a proof (e.g., when proving equivalence of two loops), several induction schemes are possible, and ACL2 includes

[1] This may be due to ACL2's avoidance of explicit quantifiers in favor of recursive functions, which are naturally dealt with using inductive methods.

heuristics for choosing which one to use. Rather than rely on these, Axe supplies explicit induction schemes for critical proofs.

Axe also uses ACL2 to manage chains of reasoning, for example when new theorems are proved by instantiating, combining, or rephrasing previously proved results. (See, for example, Section 11.9, which describes the sequence of theorems generated when characterizing a loop function; ACL2 provides a helpful check that such reasoning is performed soundly.)

Since Axe is designed to run without user intervention, the calls it makes to the ACL2 prover must generate predictable behavior. To ensure this, Axe prevents ACL2 from applying any rules or reasoning techniques other than those essential to the proof at hand[2]. Also, complicated computations and predicates are often wrapped up into functions which ACL2 is prevented from opening when it is merely putting together the pieces of a proof. Finally, chains of reasoning that include calls to ACL2 are broken into small pieces, each of which performs a single reasoning step. (This prevents ACL2 from trying to do two things at once, when those things may interfere with each other.)

10.2 Putting a Loop Into Simple Form

As described in Section 4.11, Axe's analysis of loop functions applies only to so-called "simple loop functions." This restriction significantly simplifies the process of transforming a loop function or proving a theorem about its properties. Among the requirements for a simple loop function are that it be tail-recursive[3] and that

[2] This is done with hints such as "`:do-not '(generalize eliminate-destructors)`" and the use of ACL2's `minimal-theory`.

[3] If the recursive call is wrapped in calls to `let`, `let*`, or `lambda`, the function is still considered tail-recursive.

it have a single "base case"[4] that simply returns some or all of its loop parameters (not performing any interesting computation before doing so). If Axe encounters a function not in simple form, Axe attempts to put it in simple form, using the methods described in this section.

10.2.1 Making A List Builder Tail-Recursive

Cryptographic algorithms often involve operations that build up lists one element at a time. The most natural way to express such a computation in a formal specification is often as a function that calls `cons` to add a list element to the result of a recursive call of the function. Unfortunately, such a function would not be tail-recursive and so would not be directly amenable to processing by Axe. However, Axe can automatically transform such functions into equivalent tail-recursive functions that it can handle. It is not easy to transform a general non-tail-recursive function to make it tail-recursive, but for this kind of "list builder" function, the transformation can be done mechanically.

Consider the following function from the formal specification of the RC4 stream cipher:

```
(defun bvxor-list (size x y)
  (if (endp x)
      nil
      (cons (bvxor size (car x) (car y))
            (bvxor-list size (cdr x) (cdr y)))))
```

This function takes two lists, `x` and `y`, and computes the exclusive-or of corresponding list elements (using `size` as the width of the XOR operation). It returns a list of the

[4] A "base case" is any branch of the top level if-then-else nest that does not contain a recursive call.

same length as x (which is usually of the same length as y). This function is not tail-recursive, because it calls cons after making the recursive call. However, it fits the "list builder" pattern and so can be converted to an equivalent tail-recursive function. The transformation involves adding a new loop parameter to serve as an accumulator and having each recursive call add one element to the end of the accumulator. The values passed into the recursive call for the other loop parameters remain the same. When the exit test is true, the function returns the accumulator.

When applied to bvxor-list, the transformation produces the new function:

```
(defun bvxor-list-tail (size x y acc)
  (if (endp x)
      acc
      (bvxor-list-tail size
                       (cdr x)
                       (cdr y)
                       (add-to-end (bvxor size (car x) (car y))
                                   acc))))
```

Note that the new parameter acc has been added and that the value previously consed onto the recursive call, (bvxor size (car x) (car y)) is now added to the end of acc. (The function add-to-end adds a value to the end of a list, unlike cons, which add to the front of a list. Using add-to-end is necessary, because the elements must be added to the accumulator in the correct order[5].) The values passed to the recursive call for parameters other than the accumulator are the same as in the original function; size is unchanged, and x and y are cdred. Whereas the original function returned nil when the exit test evaluated to true, the new function returns

[5] Calling add-to-end is significantly slower than calling cons, because it has to rebuild the entire list, but execution speed is a lesser concern than getting the function into the proper form to be processed by Axe.

the accumulator. Note that, unlike bvxor-list, the new function is tail-recursive.

When Axe applies the transformation just described it both generates the new function and performs a formal proof equating it with the original function. For bvxor-list, Axe proves the following theorem:

```
(defthm bvxor-list-becomes-bvxor-list-tail
  (equal (bvxor-list size x y)
         (bvxor-list-tail size x y nil)))
```

Note that the accumulator must be initialized to nil for the call of bvxor-list-tail to preserve the meaning of bvxor-list. The proof is fairly mechanical and uses the following helper lemma:

```
(defthm bvxor-list-becomes-bvxor-list-tail-helper
  (implies (true-listp acc)
           (equal (append acc (bvxor-list size x y))
                  (bvxor-list-tail size x y acc))))
```

The helper lemma is proved using induction, and the main lemma follows from instantiating the helper with acc equal to nil. To see this, note that (true-listp nil) is true and that (append nil (bvxor-list size x y)) is equal to just (bvxor-list size x y).

10.2.2 Peeling Off the Base Cases

The base case of a simple loop function must not perform any actual computation; it should just return some or all of the loop parameters (perhaps as a tuple). As described in Section 11.9, this makes it easier to prove a theorem characterizing the loop function. It is fairly straightforward to “peel off” the base case from a loop function. To do so, Axe mechanically generates a new function whose base case simply returns the appropriate loop parameters. Calling this new function, followed by performing the computation previously done by the base case, is always equivalent to calling the original function. Axe proves this property (by induction), and the resulting theorem is then used as a rewrite rule to transform all calls of the original function. Peeling off the base case in this way has the additional benefit of lifting any computation previously done in the base case up into the overarching miter, which exposes the nodes involves to the Equivalence Checker. (Perhaps some of those nodes will be found to be probably equal to other nodes in the miter.)

10.2.3 Combining Base Cases

Axe is also able to combine multiple base cases of a function to make it into a simple loop function. Functions with more than one base case can arise from the constant factor unrolling transformation described in Section 12.2.1. Given a function with multiple base case branches, Axe mechanically builds a new, equivalent function with a single base case. The exit test of the new function is the disjunction of all of the branch conditions that led to base cases in the old function. The single base case of the new function is an `if` nest with one branch corresponding to each of the base cases of the old function (which may each have involved performing different computations). Axe proves (by induction, as usual), that the new function is equivalent to the old function and generates a rewrite rule to replace the old function. The new function may still not be in simple form, because, while it has only a single base case, that base case may not merely return some of the loop parameters. Thus, peeling off is usually performed next, to finish converting the function into simple loop form.

The transformations described in this and the preceding sections usually suffice to bring a function into simple loop form[6], and Axe does so before applying any other

[6] One other transformation might be necessary: combining several recursive calls into one. This would be straightforward to implement, but the need for it has never arisen in practice.

analysis to a function. Thus, unless otherwise stated, the rest of this dissertation deals only with simple loop functions.

10.3 Traces

Many of the analyses and transformations that Axe performs on loop functions are based on the examination of execution traces. Chapter 13 describes how Axe uses traces to discover loop invariants and "connections". The lengths of traces (corresponding to the number of loop iterations performed on each test case) can indicate that a function never executes its loop body, could be completely unrolled (Section 10.4), or should be transformed by constant-factor unrolling (Section 12.2.1) or loop splitting (Section 12.2.2).

To generate traces for a loop function, Axe uses the test cases for the overarching miter in which the loop function appears.

Recall from Section 8.3 that each test case is a valuation for the input variables of the miter and that test cases can be used to assign values to the other nodes through evaluation (except that some nodes may be unused on some test cases). If a node in the miter contains a call to a loop function, that evaluation process involves calling the Axe Evaluator (described in Section 4.9) to evaluate the call to the loop function. However, when reasoning about the loop function itself, Axe needs access to the intermediate values computed during the evaluation of the loop function (the values of the loop parameters passed into each recursive call). To capture this data, Axe evaluates the loop function using its "tracing evaluator," which is identical to the Axe Evaluator except that it is instrumented to produce traces. A trace is simply a sequence of tuples, in which each tuple gives values for all the parameters of the loop[7].

[7] In general, a recursive function may contain many recursive calls, thus generating traces that are trees. However, Axe only traces simple loop functions, each of which has a single recursive call. Thus the traces can be flattened into linear sequences.

Each trace of a loop function corresponds to a single test case of the overarching the miter (one for which the node containing the loop function is not unused)[8].

For example, consider this simple loop, in which `i` counts up to `j` and which modifies an array of four 32-bit cryptographic values:

```
for (int i = 0 ; i < j ; i++) {
  // ... statements that modify the array but not i or j ...
}
```

The loop parameters are `i`, `j`, and the array. If `j` is initially 4, a trace might be:

```
(0 4 (3731157443 2436522247 2148470593 3355054412))
(1 4 ( 975926496 1447727341  296330417 3178671623))
(2 4 ( 840376080 1657051051 4203293735 1474011647))
(3 4 (3461975045 4068623870 1387895382 1551330983))
(4 4 ( 107902556 3331249930 3316310409 4003942895))
```

Here each element of the trace is a 3-tuple with values for `i`, `j`, and the array. During the trace, `i` counts up until it reaches `j`. As is typical for traces of cryptographic code, the data values in the array look random.

If `j` is initially 2, the trace would be shorter. It might be:

```
(0 2 ( 275539183 1174796661  374476702 2345540907))
(1 2 (1663351080 4227851480  594878417 2265059054))
(2 2 (4245172615 4147871629 3111519246 2971542628))
```

As this example shows, traces are hierarchical in that the components of each tuple can themselves be composite objects. In this case, the last component of each tuple is

[8] A single test case of a top-level miter can give rise to more than one trace of a loop in a nested miter. This corresponds to the fact that an inner loop may execute multiple times, once for each iteration of the outer loop that contains it.

an array; components can also be sequences and other tuples. The first tuple in each trace represents the values initially passed in to the function, and these correspond to the "old variables" of the loop, discussed in Section 11.3. The final tuple in each trace always contains values that make the loop's exit test true. As discussed in Section 11.9, one can often make a stronger claim about these final values than about values on an arbitrary iteration.

10.4 Complete Unrolling

In many cases, Axe can completely unroll a loop function. This approach is often advantageous for automated verification, because it avoids the need to discover and prove inductively strong loop invariants. The disadvantage of unrolling is that it can generate very large terms, but Chapter 8 describes how Axe deals with such terms.

Complete unrolling is the process of replacing a loop with an equivalent computation that contains multiple copies of the loop's body but lacks the original loop construct. This is only possible if there is some bound on the number of loop iterations that could possibly performed on any execution of the program.

In the simplest case, the number of iterations performed by the loop is the same for every execution. If c is that number, then the loop can be completely unrolled into a computation that involves c executions of the loop body, performed in sequence. Section 10.4.1 describes how Axe uses "opener rules" to unroll such loops.

Even if the iteration count for a loop is not the same for every execution, the count may still be bounded. In such a case, complete unrolling can still be performed. Let k be the maximum number of loop iterations performed by the loop on any execution. Then the loop can be unrolled into an expression that contains k copies of the loop body. Because not all executions will necessarily involve executing the loop body all k times, if-then-else operators must be present in the unrolled form to control how many times the loop body is actually executed. Section 10.4.2 described how Axe detects and completely unrolls this sort of loop.

Axe takes pains to ensure that complete loop unrolling is sound. The opener rules of section 10.4.1 and the transformations of section 10.4.2 are proved as ACL2 theorems, to ensure that complete unrolling preserves the meaning of the computations involved. Complete unrolling leaves a "residual" term which captures any computation beyond that performed by the unrolled loop body; eliminating this residual requires proving that the unrolling amount is in fact sufficient (that is, that no execution could ever require more executions of the loop)[9].

Unfortunately, some loops cannot be unrolled. Consider a program that takes as input a message of any length. Since the entire message must be processed, the amount of computation done by the program is not bounded. Thus, there is no way to express the computation without loops (or recursion). Other programs may have loop iteration counts that are technically bounded but very large. For example, if the input to a program were restricted to fit in an array in the Java programming language, and that array were processed one element at a time, then the main loop would technically have a bound of $2^{31}-1$ iterations (since the length of a Java array is a 32-bit signed `int`). However, this bound is not useful; the completely unrolled loop would be represented by a huge expression that included $2^{31} - 1$ copies of the loop body (and roughly the same number of if-then-else expressions). Such a representation would be too large to handle. Functions that cannot be completely unrolled must be dealt with using the inductive methods described in Chapters 11 and 12.

If the body of a loop contains inner (nested) loops, complete unrolling of the outer loop will not directly result in a loop-free term, because the unrolling process will expose the inner loops. However, the inner loops may also be unrollable. (If not,

[9] Axe can detect that the residual loop function never executes its body; whenever it finds such a function, Axe unrolls it one more step, to expose its exit test to the Equivalence Checker, which will then attempt to prove that the exit test is always true.

they must be dealt with using inductive methods.)

Complete unrolling works well in the verification of block ciphers. The amount of computation performed by a block cipher is usually bounded (often with a fixed, constant iteration count for each loop), so all the loops in a typical cipher can be completely unrolled. The usual approach to apply Axe to such programs is to perform symbolic execution (described in Section 9.3), which in essence performs complete unrolling of all loops.

However, complete unrolling of all loops is usually not possible for the stream ciphers or the cryptographic hash functions, because such algorithms typically have inputs whose length is not fixed. Luckily, "hybrid" approaches can usually be applied to these algorithms, since their outer loops typically process the input in fixed-size "blocks." The outer loop must be dealt with inductively, but the computation performed for a single block is usually completely unrollable.

The rest of this section explains the complete unrolling process in detail.

10.4.1 Complete Unrolling With Opener Rules

In some cases, Axe can perform complete unrolling using "opener rules." The obvious way to unroll a loop would be to use its definitional axiom as a rewrite rule, replacing the function call with the instantiated body. Unfortunately, this could be dangerous, because it might cause a rewriting loop. In particular, if the exit test of the function call cannot be resolved (rewritten to a constant), "opening" the function would introduce an if-then-else term containing another call to the function. That call's exit test would probably also be unresolvable, so it would open, introducing another call to the function, and so on. The use of opener rules makes such looping rewrites much less likely, by requiring that the exit test be resolvable before either rule fires.

The body of a typical recursive function is an `if` expression that uses an "exit test" to select between a "base case" in which values are returned without making a recursive call and a "recursive case" that does contain a recursive call to the function. (Of course, more complicated function bodies are possible, including bodies with more than one base case branch or more than one recursive call. However, the simple schema for a recursive function described here is by far the most common for the functions of interest to Axe. Note in particular that all "simple loop functions" fit this pattern.) For a typical recursive function, two rewrite rules (collectively called "openers") may be defined. The "base case" rule allows the function call to be replaced with the base case branch, provided that the exit test is true. The "unroller" rule allows the function call to be replaced with the recursive branch, provided that the exit test is false.

For example, consider the following simple loop function from the formal specification of the RC6 cipher:

```
(defun keyloop1 (i max s)
  (if (or (> i max)
          (not (natp i))
          (not (natp max)))
      s
      (let ((s (update-nth i (word+ (nth (+ -1 i) s) *q32*) s)))
        (keyloop1 (+ 1 i) max s))))
```

In this function, `i` counts up to `max`, and the termination test is essentially `(> i max)`. (The terms `(not (natp i))` and `(not (natp max))` are present for technical reasons to ensure the function always terminates, even when given arguments that are not natural numbers.)

The function keyloop1 gives rise to the following two opener rules:

```
(defthm keyloop1-base
  (implies (or (> i max)
               (not (natp i))
               (not (natp max)))
           (equal (keyloop1 i max s)
                  s)))


(defthm keyloop1-unroll
  (implies (not (or (> i max)
                    (not (natp i))
                    (not (natp max))))
           (equal (keyloop1 i max s)
                  (let ((s (update-nth i
                                       (word+ (nth (+ -1 i) s) *q32*)
                                       s)))
                    (keyloop1 (+ 1 i) max s)))))
```

Rule `keyloop1-base` requires the termination test to be true and replaces the call to `keyloop1` with its base case expression, which is just `s`. Rule `keyloop1-unroll` assumes the negation of the termination test and replaces the call to `keyloop1` with the recursive call on updated arguments.

The caller of `keyloop1` always calls it with a value of 1 for `i` and 43 for `max`. On each recursive call, `i` is incremented and `max` remains `43`. Rewriting the call of `keyloop1` using the opener rules proceeds as a sequence of steps, with each application of `keyloop1-unroll` generating a new call to `keyloop1` in which the loop parameters have been updated. Each unrolling step increments the value of `i`. For example, when `i` is `27` and one unrolling step is applied, the new term for `i` will be `(+ 1 27)`, which simplifies to `28`. The most important effect of unrolling is that the expression for the `s` parameter grows for each unrolling step, eventually accumulating 43 “copies” of the call to `update-nth` (each containing the appropriate expression for `i`). Unrolling occurs 43 times and stops when finally `i` is 44, at which point `keyloop1-base` fires to strip off the residual call to `keyloop1`, leaving just the large expression for the final value of `s`.

Since the variables involved in the loop exit test (`i` and `max`) are always known constants during the unrolling process, the Rewriter will always be able to relieve the hypothesis of exactly one of the two opener rules.

Complete unrolling using opener rules has several advantages compared to the approach described in the next section. It can be applied to recursive functions that are not tail-recursive, and it does not require the execution of test cases to determine the "unrolling amount" for a function. Its main drawback is that it cannot be used to unroll loops whose iteration counts are bounded but are not the same on every execution. If the iteration count of a loop function is not the same on every execution, it is unlikely that either the exit test or its negation will be resolvable. Thus, neither the unroller rule nor the base case rule will fire.

10.4.2 Complete Unrolling Using Splitting and Forced Unrolling

Axe can mechanically perform unrolling of a loop function, even when the iteration count for the function is not constant, as long as the iteration count is bounded. This approach thus works in some cases in which opener rules fail. (However, the techniques of this section only apply to simple loop functions.)

Completely unrolling a loop with a bounded iteration count involves two steps. First, Axe examines execution traces to determine the bound. (Of course, in some cases there will be no bound, which means the loop is not completely unrollable.) Second, Axe performs the actual unrolling in a mechanical way that involves splitting the loop into two pieces and forcing the first piece to unroll.

Detecting that a loop has a bounded iteration count is a bit tricky. Axe examines execution traces of the loop function, which correspond to the test cases supplied for the overarching miter. Because the set of test cases used by Axe is necessarily finite, however, every loop appears to have a bounded iteration count (the bound being the largest number of iterations performed for any test case). Of course, some loops are not actually bounded (or are technically bounded but with bounds that are too large to be useful). A second complication is that, even if the number of loop iterations is bounded, Axe might not have a test case on which the maximum number is actually achieved. For example, if the number of loop iterations is always `(mod (len input) 64)`[10], then the loop iteration count is at most 63. If the length of `input` is uniformly distributed, then this maximum will occur only once every 64 test cases (or less often if the loop is not "used" on every test case). So random generation of test cases may fail to actually generate such a test.

Axe addresses these issues by attempting to discover an expression for the number of iterations as a function of some input variable. It then checks whether the resulting expression is bounded. For example, if the iteration count is expressible as `(len input)`, then it is not bounded (unless the input length is bounded). If it is `(mod (len input) 64)`, then it is bounded, since the value of the `mod` is at most 63. Axe's ability to detect such patterns is currently a bit limited but could be generalized. Axe also allows the user to manually specify an unrolling amount for each loop function that is to be completely unrolled.

For a loop with a bounded but not constant iteration count, opener rules are unlikely to work. The process would not even get started, because the exit test of the outer function call would not be resolvable. To force unrolling based on a bound on the iteration count, Axe splits the loop into two parts. The first part is a

[10] This might occur for a loop that processes any extra bytes after all complete 64-byte blocks have been processed.

"limited" function that is identical to the original loop function except that it takes an additional argument: an explicit number of loop iterations to be performed. The limited function exits on a given iteration if either the original function would exit on that iteration, or the count of loop iterations left to perform has reached zero. It returns all of the loop variables, and these are passed into the second part of the newly split loop, which is just an alias of the original loop function[11]. The alias function performs any extra loop iterations that remain after the execution of the limited function. The key property is that the composition of these two functions returns the same value as the original loop would, regardless of the number of iterations passed in to the limited function. (If the limited function is given an iteration count that is too large, it simply exits before all of the indicated iterations have been performed, and the call to the alias function will exit immediately. If the limited function is given an iteration count that is too small, the iterations it fails to do are done by the alias function.) Axe proves this key property as an ACL2 theorem and then applies the theorem to transform the DAG. The theorem is true for any iteration count passed to the limited function, but Axe chooses the count that will cause the limited function to perform the maximum number of possible loop iterations. This is just the number that Axe believes is the upper bound for the iteration count.

Next, the limited function is forced to unroll the given number of times, using a rule similar to the unroller rules discussed in Section 10.4.1. A key difference is that this unroller rule forces the limited function to unroll, even if the exit test of the original function cannot be resolved. The unrolling is governed entirely by the iteration count passed in, which decreases at every step. Thus, the function may be forced to unroll more iterations than are actually used on some test cases[12]. If Axe has the correct bound, the unrolled version of the limited function should be sufficient

[11] It is identical but is given a different name so that it will be unaffected by the operation that replaces the original loop function.

[12] If-then-else nodes in the result will ensure the right number of iterations is always performed.

to perform all the iterations for any possible test case. This means that the second function (the alias function) should always immediately return[13], and proving that fact shows that the bound on the loop iteration count is correct. To do that proof, Axe uses a "base case" rule for the alias function with a `work-hard` hypothesis that instructs the Rewriter to expend extra effort trying to prove that the alias function returns immediately.

The complete unrolling process described in this section is performed whenever Axe finds a bound for the iteration count of a loop (unless Axe is instructed not to unroll), whenever the user explicitly instructs Axe to unroll a particular function, or when necessary to prove a merge (i.e., when one side of a miter contains a loop function which corresponds to some loop-free computation on the other side of the miter).

10.5 Loop Functions in Miters

The application of Axe to a miter that contains loop functions follows the basic process described in Chapter 8; Axe rewrites the miter and then runs test cases and performs sweeping and merging. The main difference is what happens when Axe attempts to merge two probably-equal nodes (or prove a probably-constant node) when the nodes in question are supported by loop functions. Recall that for pure miters, merges are proved by calling STP on heuristically cut goals. However, when loop functions are present, proving the merge may require proving inductive properties of loop functions and then calling the Axe Prover, which, unlike STP, can make use of such properties.

Since impure miters generally have fewer nodes than pure miters (due to loop functions not being unrolled), Axe works harder to try to prove each merge. First it forms the equality of the nodes to be merged and rewrites it. The rewriting process

[13] We call it a "residual" of the unrolling process.

may use properties already proved about loop functions. If the equality rewrites to true, the proof succeeds. If it rewrites to `nil`, the proof fails. Otherwise, Axe analyzes the loop functions supporting the nodes in question, possibly proving lemmas about them or transforming them. Finally it calls the Axe Prover to try to prove the merge.

Axe's analysis of loop functions has several steps. First, Axe ensures that a lemma has been proved about each individual loop function (as described in Chapter 11).[14] Next, Axe detects miter nodes that support both of the nodes to be merged. These "shared supporters" correspond to the part of the DAG that has already been merged, and so Axe concentrates on loop functions above the shared supporters (but below the nodes being merged). If neither node is supported by a loop function above the shared supporters, Axe's analysis of loop functions ends (and Axe moves on to call the Prover to attempt to prove the merge). If only one node has recursive functions above the shared supporters, Axe attempts to completely unroll those functions (since they apparently correspond to loop-free code on the other side of the miter). Otherwise, Axe moves on to check for "producer/consumer" patterns that can be "streamified," as described in Section 12.2.3. Axe next checks whether any loop function should be split or unrolled by a constant factor (as described in Sections 12.2.2 and 12.2.1)[15]. Finally, Axe tries to prove a connection lemma that relates two loop functions, one on each side of the miter. This process is the central operation in the equivalence checking of programs with loops and is described in detail in Chapter 11.

If loop analysis yields no new lemmas about loop functions and performs no transformations, the Axe Prover is called as a last resort attempt to prove the merge. The Prover may use previously-proved loop properties during rewriting, in addition to

[14] Whenever a new lemma is proved, Axe abandons the current sweep and uses the new lemma to rewrite the entire miter. It then restarts the sweeping and merging process. Such restarting is also done if Axe applies a transformation to a loop function or proves a connection between two loop functions. The idea is to apply any rewrites once to the entire miter instead of repeatedly when attempting to prove many merges.

[15] These transformations currently rely on user annotations, but it should be possible to detect when to apply them by analyzing traces.

applying its usual proof strategies, such as calling STP.

The proof of a merge uses the assumptions of the miter and also uses an approximation of the "context" of the node to be replaced, as described in the next section.

10.5.1 Contexts

Recall that DAG nodes are not necessarily "used" for every test case. When Axe attempts to prove the equality of two nodes, A and B, with the intention that the merge will replace B, it is sound to assume that B is actually used. (For any test case on which B is not used, replacing it with A cannot have any effect.) How can we generate a condition that is guaranteed to hold whenever B is used, so that Axe can assume it during the proof of the merge?

Recall from Section 5.16 that if-then-else operators above a subterm provide a "context" that can be assumed when rewriting the subterm. The situation for a DAG is similar but more complicated. In a DAG, there may be exponentially many such "contexts," each corresponding to one path through if-then-else expressions from the root of the DAG to the node in question. The context provided by any one path is the conjunction of all of the if-then-else conditions along the path, with each one possibly negated (depending on whether the path includes the then-branch or the else-branch). The "precise context" of the node is the disjunction of the path conditions for all paths to the node. Whenever the precise context of the node is true on a test case, the node is "used," and whenever the precise context is false, the node is unused. Thus, it would be sound to assume the precise context of B when doing the proof to replace B by A.

However, the precise context of a node may be too large to handle. In the worst case, there may be exponentially many paths to a node[16], yielding a precise context

[16] in terms of the number of if-then-else operators above it in the DAG

with exponentially many disjuncts. Thus, Axe actually computes an approximation of the context. The approximate context is implied by the precise context and is thus safe to assume for the proof. (If the approximate context is false for any test case, then so is the precise context, making the replaced node irrelevant for that test case.) Of course, the approximate context may be weaker than the precise context.

To compute the approximate context for a node (hereafter we will just call this the "context" of the node), Axe computes the contexts of all nodes above it in the DAG. The context computed for a node is always a conjunction (of other nodes representing boolean terms) and is logically implied by the precise context of the node. The restriction to use conjunctive form loses information. In particular, when the context of a node is computed from the contexts of its parents (assuming for now that the parents are not if-then-else operators), we would like to simply use the disjunction of all of the parent contexts. Since that result cannot be expressed as a conjunction, Axe has to do something else. It takes the intersection of the facts in the parents contexts. For example, if a node has two parents, one whose context includes conjuncts A and B and one whose context includes conjuncts A and C, the context for the node itself will include only A. The context for a node is safe to assume but loses information (such as the fact that either B or C must be true for the node to be used). (Another source of information loss is that the procedure does not really consider the meanings of the context facts or the relationships between them. For example, if B and C were known to be equivalent, the context just mentioned could legally contain both A and B, but Axe would not realize this unless B and C were actually the same node.)

Nodes whose parents are if-then-else operators have information added to their conjunctions when appropriate. For example, if a node is the then-branch of an `if` node with condition C, and if all other parents of the node also have condition C in their contexts, then C is included in the context for the node. (Axe takes pains to handle tricky cases, such as when a node is both the then-branch and the else-branch of the same parent node.) The effect of Axe's algorithm is that the context of a node will include any facts tested by if-then-else operators on every path to the node. This is relatively cheap to compute and is usually sufficient for the proofs performed by Axe. (In the worst case, if more precise contextual information is required, the proof may have to wait until Axe eventually performs case splits of the miter using the necessary facts.)

The algorithm's treatment of if-then-else conditions does exploit some of the boolean structure of those conditions. For example, if a node is the then-branch of an `if` whose test is `(booland a b)`, the computed context would include both `a` and `b` separately, rather than the `booland` fact. Likewise, for a node that is the else-branch of an `if` whose test is `(boolor c d)` the context will use both `(not c)` and `(not d)` rather than `(not (boolor c d))`. (This process of harvesting disjuncts and conjuncts from boolean terms is similar to the operation performed when "promoting" disjuncts in the Axe Prover, described in Section 7.2.)

If the context for a node is contradictory (because it contains both some fact and its negation), then the node is irrelevant in the DAG, and the proof done to justify replacing the node succeeds immediately.

Chapter 11

Proofs and Transformations for a Single Loop Function

Whenever a loop function cannot be completely unrolled, Axe must handle it using inductive techniques. The four main strategies that Axe employs are:

1. Detecting and proving properties of a single loop function.
2. Transforming a single loop function to make it simpler or more regular (e.g., dropping redundant parameters or "uncdring").
3. Detecting and proving an inductive connection relation between two loop functions.
4. Transforming a loop function to make it match up with a loop function on the other side of the miter (e.g., the " synchronizing transformations" of loop splitting and constant factor unrolling).

This chapter describes the first two of these topics, which cover how Axe deals with a single loop function in isolation. The main analysis performed on a loop function involves formulating and proving an inductive invariant about it and then using the invariant to generate a theorem characterizing the value returned by the loop function. However, Axe may first transform the function, in order to simplify it or make it easier to reason about. Axe uses both static and dynamic analysis to decide when to apply such transformations; the need for some transformations is clear from analysis of the syntactic structure of a loop function, but others require the examination of execution traces.

11.1 The "Uncdring" Transformation

"Uncdring" is a transformation that applies to certain list processing functions that often appear in Axe's formal specifications. It can make such a function more similar to an analogous function derived from imperative code.

A common practice in ACL2 (and Lisp programming in general) is to process the elements of a list by "`cdr`ing down" it. A function written in this way would process the first element of the list, discard that element by calling the function `cdr` on the list, and then call itself recursively on the resulting, shorter list. Such a scheme is simple, natural and efficient, and it is commonly used in the formal specifications discussed in Section 4.12.

Consider the following simple example:

```
(defun sum-list-aux (lst acc)
  (if (endp lst) ;endp recognizes an empty list
      acc
      (sum-list-aux (cdr lst) (bvplus 32 (car lst) acc))))

(defun sum-list (lst)
  (sum-list-aux lst 0))
```

Here the function `sum-list` returns the (32-bit) sum of the elements in its input, which can be of any length. It initializes an accumulator to 0 and calls the helper function sum-list-aux, which "cdrs down" its lst parameter, updating the accumulator as it goes. Note that as sum-list-aux executes, the value of the lst parameter gets shorter as elements are discarded. (The function returns when lst is empty.)

Now consider how the same computation would be written in Java, with the sequence stored in an array:

```
public static int sumList(int[] in) {
   int sum = 0;
   for (int i = 0; i < in.length ; i++)
      sum += in[i];
   return sum;
   }
```

Here, successive elements of in are accessed using the expression in[i]. The crucial thing is that the array, in, remains unchanged throughout the execution. Now imagine how the equivalence proof of these two implementations would go. The full details of such a proof are discussed in Section 12.1, but a key step is to find a relationship between the variables of the two loops. For the sequence variables, that relationship would be a bit tricky, because sum-list-aux cdrs down lst, but sumList leaves in unchanged. One could express the relationship using the function nthcdr (which calls cdr repeatedly), as (equal lst (nthcdr i in)[1]. But reasoning about claims that include sequence functions like nthcdr is tricky, in part because such functions are outside of the language of STP. Furthermore, experience with Axe has shown that automating such reasoning can be especially difficult when other differences between the sequences are involved (e.g, if one sequence has been "packed" or "padded").

The uncdring transformation provides a way to avoid this whole issue. To apply

[1] Here, we get a bit lucky, because the number of cdrs to perform can be expressed simply as i.

uncdring, Axe identifies a parameter that is `cdr`ed down and generates a new, equivalent function in which that parameter is unchanged but has every use surrounded by an appropriate call of `nthcdr`. A new parameter is added to track the number of `cdr`s to be performed by `nthcdr`. For example, applying uncdring to `sum-list-aux` yields:

```
(defun sum-list-aux-uncdred (numcdrs lst acc)
  (if (or (not (natp numcdrs)) ;included for technical reasons
          (endp (nthcdr numcdrs lst)))
      acc
      (sum-list-aux-uncdred (+ 1 numcdrs)
                            lst
                            (bvplus 32 (car (nthcdr numcdrs lst)) acc))))
```

Here, the parameter `lst` is passed unchanged to the recursive call, but a new parameter, `numcdrs`, has been added (and is incremented on each iteration). To preserve the behavior of the function, every reference to `lst` has been replaced by `(nthcdr numcdrs lst)`. The new function `sum-list-aux-uncdred`, when called with `numcdrs` equal to `0`, is equivalent to the original function.

Does the inclusion of `nthcdr` in the new function cause problems? It turns out that having `nthcdr` in a function definition is better than having to mention it in a loop property, because its occurrences tend to be inside calls to the functions `car` and `endp` (as in the definition of `sum-list-aux-uncdred`). The pattern `(car (nthcdr numcdrs lst))` can be handled by rewriting it to `(nth numcdrs lst)`, and the pattern `(endp (nthcdr numcdrs lst))` can be rewritten to a claim comparing `numcdrs` and the length of `lst`. The new parameter, `numcdrs`, does not complicate things too much, because it is just a number, and numbers are easier to reason about than sequences.

In keeping with Axe's philosophy of checking the transformations that it performs, the application of uncdring is not simply assumed to be correct. Instead, Axe performs a formal proof. For our example, the theorem is:

```
(defthm sum-list-aux-becomes-sum-list-aux-uncdred
  (equal (sum-list-aux lst acc)
         (sum-list-aux-uncdred 0 lst acc)))
```

The proof is straightforward enough that it can be completely automated. The resulting theorem is used as a rewrite rule to replace all mentions of the old function with its uncdred version.

Applying uncdring simplifies the later comparison of the loop function with an equivalent loop. For this example, the connection between the `lst` parameter of `sum-list-aux-uncdred` and the `in` parameter of the Java program `sumList` is simple; they are equal. Thus, applying uncdring simplifies the reasoning that will be needed to connect the two functions.

If more than one parameter of a function is `cdr`ed down, uncdring applies to all of the `cdr`ed parameters simultaneously and introduces a single new `numcdrs` parameter which is used for each uncdred sequence.

11.2 Proving A Theorem About A Loop

When no loop transformations apply, Axe attempts to characterize the execution of the loop function by proving a theorem about the values that it returns[2]. If the function models a loop in an imperative program, these properties correspond to facts about the program variables after the execution of the loop. Such properties are helpful for several reasons. First, they may provide critical information that enables simplifications in the DAG above the loop function. For example, if the loop function returns a value that is later used as an array index, it may be critical to know that

[2] A function may return more than one value by grouping them into a tuple, which is technically a single value. When we talk about a function's "return values," we mean the components of such a tuple.

the value returned is within a certain range. Likewise, it may be important to know the types of the values returned by the loop function.

Also, some of the values returned by the loop may be expressible in a simple "closed form" that does not involve the loop function. For example, if a loop index increases until it reaches some constant maximum value, then the final value of the index will simply be that constant. Proving this fact would allow the corresponding return value of the loop function to simply be replaced with the constant. (Also, any value that is just passed through the loop and returned unchanged is trivially expressible in closed form.)

In order to prove strong properties about a loop function, Axe needs to discover and prove an inductively-strong loop invariant for the function. The loop invariant is a formula over the parameters of the loop function (and perhaps their corresponding "old variables," as described in the next section). The loop invariant must hold on the initial values passed into the loop function and on the updated values passed to every recursive call (including the final call, for which the exit test will evaluate to true and the loop body will not actually be executed). If the loop function represents a loop in imperative code, the loop invariant should be true every time control reaches the "top" of the loop.

It may happen that the loop invariant for a function allows some loop variables to be expressed in terms of others. If so, Axe may use such facts to simplify the function by eliminating the redundant loop variables. Otherwise, Axe proceeds to use the loop invariant to prove a theorem about the values returned by the loop function.

As described in Chapter 12, the loop invariants proved for individual loop functions provide the starting points for proofs of connections between equivalent loop functions.

11.3 "Old Variables"

Loop invariants are predicates over the variables of loop functions. However, they may also mention "old variables" that represent the initial values of the loop variables (the values passed into the outermost calls of the functions). For example, the loop invariant `(equal x oldx)` indicates that the value of the loop variable `x` does not change during the execution of the loop[3]. This is called an "unchanged variable invariant." Such invariants are detected by Axe (Section 13.2) and can be used to "push back" other invariants, so that they mention the old versions of any unchanged variables. For example, if `x` is unchanged during the execution of a loop function, then any invariant mentioning `x` could be changed to instead mention `oldx`. This is often desirable, because the value initially passed into the function is expressible without mentioning the loop function. For example, consider a loop function, `foo`, which takes one parameter, `params`, for which component 3 is unchanged. This fact will be expressed as a rewrite rule that turns `(nth '3 (foo params))` into simply `(nth '3 params)`. If component 3 is mentioned in other rules about `foo`, those mentions should be pushed back, because the `(nth '3 (foo params))` pattern may be simplified away before the other rules can be applied.

Old variables can also express facts about monotonic sequences. For example, the invariant `(not (bvlt '32 x oldx))` uses an old variable to express the fact that the 32-bit (unsigned) loop variable `x` does not decrease during the execution of the loop. More complicated properties involving old variables are possible, especially in expressions which allow "redundant" loop variables to be expressed in terms of other loop variables (see Section 11.7).

The use of old variables is not unique to Axe. For example, they are used in JML,

[3] Throughout this discussion we will prepend `old` to the name of a loop variable to indicate its corresponding "old variable." In practice, Axe generates unique names for old variables to avoid name clashes (e.g., with a parameter already named `oldx`). Old variables are indicated unambiguously in user-provided annotations by wrapping the variable in a call of the pseudo-function `oldval`.

the Java Modeling Language [39][4].

11.4 The Use of Traces

Section 10.3 discussed how traces for loop functions in a miter can be generated using test cases for the miter. Such traces are used in two ways when proving theorems about a loop function. First, analysis of traces is used to discover loop invariants. Second, traces provide test cases for the "nested" miters used to reason about the bodies of loop functions.

In order to analyze a loop function, Axe requires at least two traces of its execution. If no traces exist, it is likely that the loop function's node is irrelevant in the overarching miter due to a contradictory context imposed by the if-then-else nodes that govern it. In this case, Axe ignores the loop function. If exactly one trace exists, analysis is also skipped, because a single trace is not sufficient to reliably detect properties that are true of every loop iteration. In this case, the user should rerun Axe using more test cases for the top-level miter. (A more sophisticated approach, not necessary for the examples discussed in this thesis, would be to try to find interesting test cases that exercise all the conditionals and nested loops in the miter.)

11.5 Finding Inductively Strong Invariants

When Axe proves an invariant for a loop function, it relies on a set of candidate invariants detected by analyzing execution traces of the function. The generation of candidates is described in detail in Chapter 13. The invariant detection process,

[4] A common question in a language with an "old" operator is whether it can be applied to entire expressions. In JML it can, so `old(1+x)` is equivalent to `1+old(x)`. However, `old(foo).bar` may be different from `old(foo.bar)`, because the latter uses the old value of the implicit Java heap. In Axe, all variables in a term are explicit, so `oldval` applies only to variables and it makes sense to talk about "old variables," rather than "old expressions."

even for nested loops, uses traces that correspond to real test cases for the outermost miter (i.e., real inputs to the programs being proved equivalent). Axe only generates candidate invariants that hold on all of the execution traces that it analyzes, but of course this does not constitute a proof that the invariants are always true. It may be that other test cases would violate some of them. Thus, Axe must find a set of invariants that it can actually prove. The set must be inductively strong; that is, if we are allowed to assume that all of the invariants in the set hold before a loop iteration, we must be able to prove that they all continue to hold after the loop iteration (assuming the loop does not exit). Axe finds the maximal set of candidate invariants that is "inductive" in this sense[5].

The candidate invariants in a set under consideration are implicitly conjoined; Axe does not consider more complicated boolean combinations of candidate invariants. This does not seem to be a severe restriction in practice.

To check whether a set of candidate invariants is inductive, Axe forms a miter and calls the Axe Equivalence Checker (Chapter 8) to prove it. If nested loops are present, the Equivalence Checker handles them in the usual way. Thus, the operation of analyzing a recursive function is mutually recursive with the operation of proving a miter; proving a miter requires analyzing the loop functions it contains, and analyzing loop functions requires proving miters about loop bodies – which may themselves contain loops to be analyzed, and so on. The depth of the recursion depends on the depth of nested loops in the program being analyzed.

[5] Because Axe does not have complete decision procedures for some of the operations that may appear in invariants (such as those involving sequences of unknown length), it may miss a set of invariants that is actually "inductive" but that it can't prove. The proof attempt might also time out. However, for the examples discussed in this dissertation, Axe doesn't fail to prove any critical invariants.

11.5.1 Finding the Maximal Inductive Set of Invariants

Axe deals gracefully with failures to prove sets of invariants. In particular, it finds the maximal subset of the candidate invariants that is actually provable. First, Axe tries to prove the entire set of candidates, as described above. If the proof succeeds, then Axe has clearly found the maximal inductive set of candidates. If the proof fails, Axe determines which of the candidate invariants failed to prove, by trying them one at a time. (Each of these proofs still assumes that all of the candidates hold before the loop body.) Any candidates that fail to prove are discarded. This is appropriate, because they could not be proven after the loop body even when assuming a large set of invariants before the loop body. Since further attempts will deal with smaller sets of invariants, it is unlikely that a discarded invariant would suddenly become provable at a later step of the process.

After discarding the invariants that fail to prove, Axe attempts to prove the conjunction of the entire remaining set. If that fails, it again determines which individual conjuncts failed, discards them, and tries again with the remaining set. This process continues until an inductive set of invariants is found. (If all candidate invariants are discarded, analysis of the loop function fails, but this rarely happens. Usually there are at least some weak type properties that are inductive.)

As mentioned above, Axe can deal with invariants that fail to prove, but this failure does have a cost. Any unprovable candidates in a set invalidate all of the work done to prove other members of that set, since the proof process may have relied on assumptions that turned out not to be provable. This penalty is especially acute when nested loops are present, in which case the proof of a set of invariants may involve significant time spent analyzing nested loops (either by completely unrolling them or proving inductive properties of them). Thus, it pays not to be wrong too often about invariants.

11.5.2 User-supplied Invariants

Axe can automatically discover many loop invariants by examining traces, but it also accepts user-supplied invariants. If any of the user invariants fails to hold on any trace, Axe signals an error. Otherwise user invariants are added to the set of candidate invariants before the proof attempts begin. The user can also specify that certain invariants should be removed from the set of candidates, either so that Axe doesn't spend time proving them, or to prevent the final theorem for a loop function from containing a precondition that is difficult to establish.

While Axe can discover many invariants automatically, and while it is fairly easy to extend Axe's invariant discovery code to new types of invariants, no tool could automatically generate every invariant that might ever be required for program verification. User annotations thus provide a way to use the Axe framework for programs with subtle invariants[6]. Of course, "user annotations" need not actually be generated by a human; invariants discovered by other program analysis tools could be fed into Axe as user annotations. (This might provide an interesting avenue for future work.)

11.5.3 Generating Real Test Cases For Loop Bodies

Recall that the process of proving a miter depends on test cases (to find probably-equal nodes and to discover properties of loops). Thus, when Axe uses the Equivalence Checker to attempt to prove that a loop body preserves some invariants, it needs a set of test cases representing real executions of the loop body. These test cases can be derived from the traces of the loop function. Recall that each trace captures a sequence of function calls constituting one complete execution of the loop. The parameters of each call in the trace are updated by the loop body to compute the parameters for the next call, except for the parameters of the last call, which are not

[6] The proofs of MD5 and SHA-1 for arbitrary message lengths currently require some user-supplied invariants, but they could perhaps be automated.

updated, since the loop exits. The test cases for the loop body thus correspond to the elements of the execution traces, excluding the last element of each trace.

The loop in Section 10.3 had two traces, of four and two loop iterations respectively, and these would yield a set of six test cases for the loop body, namely:

```
((0 4 (3731157443 2436522247 2148470593 3355054412))
 (1 4 ( 975926496 1447727341  296330417 3178671623))
 (2 4 ( 840376080 1657051051 4203293735 1474011647))
 (3 4 (3461975045 4068623870 1387895382 1551330983))
 (0 2 ( 275539183 1174796661  374476702 2345540907))
 (1 2 (1663351080 4227851480  594878417 2265059054)))
```

Note that the six test cases are ordered so that all of the test cases derived from the first trace (for which `j` is 4) come first. This can be a problem when the test cases are used in mitering, because, as described in Section 8.3.2, Axe can stop the process of finding probable facts if many tests have been analyzed without any new information being discovered. If Axe stopped running tests before the fifth one listed above, it would incorrectly conclude that `j` is always 4 (and might waste time trying and failing to prove that fact)[7]. In order to ensure that a wide variety of behaviors is seen early in the search for probable facts, Axe shuffles the order of the test cases. So the list of test cases actually passed to the Equivalence Checker might actually be, say,

```
((2 4 ( 840376080 1657051051 4203293735 1474011647))
 (1 2 (1663351080 4227851480  594878417 2265059054))
 (0 4 (3731157443 2436522247 2148470593 3355054412))
 (0 2 ( 275539183 1174796661  374476702 2345540907))
```

[7] Of course, Axe would never stop after just four test cases, but long traces may give rise to many test cases, and the first few traces analyzed may not capture all of the possible behaviors.

```
 (1 4 ( 975926496 1447727341  296330417 3178671623))
 (3 4 (3461975045 4068623870 1387895382 1551330983)))
```

11.6 Using the Invariants

After finding an inductive set of invariants, Axe checks whether it contains any explanations of loop variables. An explanation is a claim that equates a loop variable with a constant or with some term involving other loop variables. If such a claim always holds, the function can be simplified by eliminating the redundant loop variable (since it can always be computed from other loop variables). The process by which Axe generates an equivalent but less redundant function is described in the next section.

If there are no redundant variables (or if eliminating them is considered unfavorable or disallowed by the user), Axe uses the set of inductive invariants to prove a theorem about the values returned by the loop function. Axe first forms the conjunction of the proved invariant candidates; this is referred to simply as the invariant of the loop. Axe proves that the updates performed by the loop body preserve this invariant, assuming the negation of the loop exit test[8]. This theorem is essentially the inductive step (the difficult part) of the induction proof performed to characterize a loop function. However, the invariant is "improved," as described in Section 11.8, before it is used to build the final theorem proved about a loop function (Section 11.9).

[8] The proof follows easily from the theorems for the individual conjuncts.

11.7 Dropping Redundant Loop Variables

As mentioned in the previous section, Axe can simplify a loop function by eliminating loop variables that can be explained in terms of other loop variables (or are constant)[9]. Quite often, the redundant loop variable is always equal to a particular constant. For example, when analyzing a particular loop in the Java implementation of SHA-1, Axe discovers the explanation:

```
(equal (nth ’10 params) ’4)
```

It turns out that loop parameter 10 represents the length of some buffer, but that value is always 4, and Axe can prove that fact and then use it simplify the function by just hard-coding 4 into the loop body and eliminating the parameter. Likewise, Axe can prove the explanation:

```
(equal (nth ’11 params) ’t_byte)
```

for the same loop. This represents the type of an array, which it turns out is always an array of bytes[10].

For the same loop, Axe detects explanations of some loop variables in terms of others, including:

```
(equal (nth ’1 params) (nth ’9 params))
```

This represents the fact that the index of the next byte to be read from the current input array is always equal to the total number if bytes processed so far, because the data is sent into the algorithm in a single chunk. Since the variables are always equal, it is not necessary to keep both. Axe also detects:

[9] This is similar to Hu and Dill’s technique of eliminating “functionally dependent variables” to reduce the sizes of BDDs [31].

[10] This type is passed around due to an idiosyncrasy of Java, in which the same bytecodes (`baload` and `bastore`) are used for both arrays of bytes and arrays of booleans. The details of the operations (how values are sign-extended) depend on whether the values stored are bytes or booleans.

```
(equal (nth '12 params)
       (len (nth '13 params)))
```

Here parameter 12 is always just the length of parameter 13 (an array), so it is not necessary to keep both[11]. Axe also finds:

```
(equal (nth '3 params)
       (slice '5 '2 (nth '1 params)))
```

Here parameter 3 is an index into a temporary buffer of sixteen 32-bit words[12], and parameter 1 is the index of the next input byte to read. As four bytes of input are consumed, the index of the buffer advances by 1 (because the four bytes are packed into a single word), so the buffer index can be found by discarding the low two bits of the index for the input array. (The slice from bit 5 down to bit 2 is makes sense because four bits are needed to represent an index into a 16-element buffer.) Since the index into the buffer can always be calculated from the input index, there is no point in passing it around. Of course, Axe doesn't actually go through this reasoning; it just looks for explanations of values as slices of other values and then tries to prove those facts. The final explanation Axe finds for this loop is:

```
(equal (nth '0 params)
       (bvminus '32
                (nth '12 params)
                (nth '1 params)))
```

This indicates that the number of bytes left to be processed in the current input chunk is the length of the current input chunk minus the total number of bytes processed so far (again, because all of the data is passed in as a single chunk).

[11] The bytecode decompiler could perhaps be changed to not generate separate loop variables for both an array and its length.

[12] Actually, it is an array of 80 words, but only the first sixteen are in use at that point in the code.

In this example, Axe is able to automatically detect and prove that six of the fourteen loop variables are redundant. This lets it build a considerably simpler function with only eight parameters.

Having a simpler function pays off when a loop is later compared to an equivalent loop from another implementation, because fewer connections will be detected between the loops. Consider two loops, each with a pair of linearly related (and, thus, redundant) loop variables. If any linear relationship exists between a variable in one loop and a variable in the other, then four total relationships will hold (each of the variables in the first loop will be related to each of two variables in the second loop). If, however, one variable in each loop is first dropped, only a single connection relationship will hold between the loops.

The details of how Axe finds explanations are described in Section 13.6, but one aspect of that process is relevant here: Axe avoids generating circular explanations. That is, it avoids generating a set of explanations such that a variable can be explained in terms of itself, by following a chain of explanations.

Given the set of explanations that ended up being provable (if any), Axe systematically builds a new, simpler version of the loop function in which redundant parameters are dropped. In particular, it builds a new base case, exit test, and loop "body" (consisting of the update functions for each parameter). Essentially, every mention of a dropped variable is replaced by its expression in terms of the remaining variables[13]. Axe proves the equivalence of the base cases, the exit tests, and the loop bodies of the two functions. The proofs assume the function's proven loop invariant, including of course the explanations.

[13] This description glosses over some details, such as the renumbering that must be done if components of tuples are dropped, and the fact that expressions are simplified after the elimination of dropped variables.

Axe then performs an ACL2 induction proof of the equivalence of the two functions[14]. This theorem states that a call of the original function can be replaced by a call of the new function, on appropriate parameters, assuming the invariant holds. It is used as a rewrite rule to replace the old function with the new[15].

It is possible for an explanation to mention an old variable. For example, it may be that `i` is always `j - oldj`. If `i` were dropped from the function, the new function would need a way to access `oldj`. This can be accomplished by adding `oldj` as a loop parameter when dropping `i`. Since doing so replaces a variable that changes every iteration (`i`) with one that remains unchanged (`oldj`,) it is considered to be an improvement. (Of course, Axe doesn't add a parameter for the old version of a variable if the variable is an "unchanged variable".)

11.8 Improving The Invariant

If Axe does not drop parameters from a loop function, it uses the loop invariant to prove a lemma about the values returned by the function. However, it first "improves" the inductive invariant. To do so, it repeatedly selects one conjunct of the invariant and rewrites it while assuming all of the other conjuncts, replacing the original conjunct with its simplified version. The process continues until no more changes can be made. The result is an invariant that is equivalent to the original inductive invariant but which is better in several ways. First, conjuncts that are implied by other conjuncts are removed. This can happen when there are two candidate invariants, one stronger than the other, and they both prove to be inductive. Consider the invariants

[14] The induction scheme is given explicitly by Axe and is derived from the bodies of the new and old functions.

[15] Because the base case of the new function may return loop variables that have been dropped (and thus would contain expressions to compute them from the remaining variables), the new function may not be a simple loop function. If so, the base case will need to be peeled off, as described in Section 10.2.2. This is done immediately after the new function is created.

(unsigned-byte-p '32 x) and (unsigned-byte-p '31 x). The first one says that x fits in 32 bits; it is likely to be easy to to prove if x is manipulated using 32-bit operations. The second invariant implies the first but is stronger; it also asserts that the top bit of x is 0 (thus, x represents a non-negative number in the two's-complement scheme). Because invariant detection does not know which invariants will turn out to be inductive, it may generate both of these. If they are both inductive, it can be helpful to drop the weaker one, keeping only the stronger. This is done during the invariant improvement process. In particular, the rewrite rule:

```
(defthm unsigned-byte-p-longer
  (implies (and (unsigned-byte-p free x)
                (<= free n)
                (natp free)
                (natp n))
           (equal (unsigned-byte-p n x)
                  t)))
```

would rewrite (unsigned-byte-p '32 x) to true, given the assumption of (unsigned-byte-p '31 x). Anything that rewrites to "true" during the invariant improvement process is dropped (because "true" can be dropped from a conjunction without changing its meaning). Because the conjuncts of the invariant will ultimately appear as hypotheses in a rewrite rule about the loop function, dropping conjuncts when possible means that fewer hypotheses will later have to be relieved.

Improving the invariant can also transform conjuncts into nicer forms. Consider the invariant (sbvlt '32 x y). Recall that sbvlt is the signed two's-complement comparison operator, which involves checking the sign bits of the values being compared. In many cases, values are known to be non-negative, meaning the sbvlt operation can be replaced with the simpler (bvlt '31 x y). This will happen by the application of the rewrite rule `sbvlt-becomes-bvlt` during invariant improvement.

Third, invariant improvement uses “unchanged variable invariants” to push back the other invariants, as described in Section 11.3.

Axe proves a theorem to justify the correctness of the invariant improvement process. It uses the theorems produced by the rewriting the individual conjuncts and chains them together to produce a theorem equating the original invariant and its final improved form[16]. The final theorem says that the improved invariant is equivalent to the original invariant, so it is also inductive. For the rest of this chapter,“the invariant” and “the invariants” of a loop, will be taken to mean the improved invariant and its conjuncts, respectively.

11.9 The Theorem About a Loop Function

Axe generates and proves a sequence of theorems about each loop function it analyzes. The process is quite complicated, but the most important step performs an induction proof of a theorem characterizing the loop. That proof depends on the loop invariant being inductive. Because Axe’s analysis only applies to simple loop functions, it can be highly automated. Also, as it generates the sequence of theorems, Axe uses ACL2 to manage the bookkeeping details, ensuring that each theorem follows from the previous ones (via instantiation, modus ponens, opening or closing of function definitions, etc.).

An ACL2 theorem about a function expresses some claim about the values finally returned by that function. However, a loop invariant characterizes the values of the parameters passed *in* to every recursive call. In particular, the loop invariant will be satisfied by the values passed in to the final recursive call, but the function may

[16] The theorems produced by the Rewriter are not formally proved in ACL2; they depend on the soundness of the Rewriter. But ACL2 ensures that the bookkeeping work done when improving, replacing, and possibly dropping invariants is correct.

perform some last bit of work on those values before exiting (such as, say, multiplying them all together, and returning only the product). That last bit of work would make it hard to use the invariant to build a claim about the return value. However, the final iteration of a simple loop function can do nothing but return a loop parameter or a tuple of loop parameters. Thus, the loop invariant essentially holds on the return values as well. (It may need to be modified slightly to reflect the tuple "shape" of the return value. Also, if any parameter is not returned by the function, all conjuncts of the loop invariant that mention that parameter will have to be dropped[17].)

Furthermore, sometimes a stronger claim can be made about the return values than that expressed by the loop invariant. While the invariant must hold on the values passed into every recursive call, we have additional information about the values on the final call: they must satisfy the function's exit test. For example, if the invariant asserts that $i <= j$, and the exit test is $j <= i$, then upon exit we know the stronger fact that $i = j$. Axe detects claims about values returned by a loop function by analyzing the final values of its execution traces, as described in Section 13.7. The result is a set of candidate claims about the final values of the loop. Axe then attempts to prove that these claims are implied by the invariant and the exit test. The proofs are done using the Axe Prover, and any claims that fail to prove are discarded[18].

Axe uses the loop function's invariants, and the additional "final claims" just described, to build the main claim about the return values of the function (accounting for any difference in return value shape). In doing so, it discards any claims that mention only old variables, because old variables are not returned by the function[19].

[17] Those claims may have been helpful in proving the inductive invariant, but there is no way to characterize the final value of a loop parameter that is not returned, nor should the caller of the function care about such a value.

[18] Axe also allows the user to provide candidate claims about the final values. Axe doesn't trust that these claims are correct; it evaluates them over the execution traces, throwing an error if any fail to hold, and then adds them to the set of candidate claims that it will try to prove.

[19] Again, such claims may have been helpful in proving the invariant.

The main theorem about the loop function makes claims about the return values of the function and requires that the values passed into the function satisfy the function's invariant. The invariant thus becomes a precondition of the function call. The proof is done using ACL2's support for proof by induction. (The induction scheme is always the obvious one suggested by the loop function.) The induction proof just puts together the pieces that have already been proved, namely, that the invariant guarantees that the final claims hold when the function returns (the "base case" of the proof) and that the invariant is preserved by the loop body (the "inductive case" of the proof). The resulting theorem can be used as a rewrite rule when reasoning about code that calls the loop function.

One complication remains: the theorem proved about a loop function may still mention "old variables," because its hypotheses and conclusion were built from its invariant, which can mention old variables. The old variables are simply free variables in the theorem (implicitly universally quantified, as always). Thus, they can be soundly instantiated in any way. Axe makes the obvious choice of instantiating the old variables with their corresponding non-old variables and generates a new theorem. This allows the affected hypotheses of the new theorem to be more easily relieved. For example, a hypothesis such as `(equal x oldx)`, representing the fact that `x` never changes during the loop, will be instantiated to `(equal x x)`. This is trivially true and reflects the fact that `x` has not changed when it is initially passed into the function. Likewise, if any claim about the return values of the function mentions `oldx`, those mentions will be replaced with mentions of `x`. In the final theorem about the function, `x` in some sense plays the role once played by `oldx` in the invariant; it represents the value passed into the outer call of the function.

The final theorem about a loop function includes calls to `work-hard` around its individual hypotheses. This instructs Axe to expend extra effort in attempting to show that the preconditions of the function are satisfied[20].

After a new lemma is proved about a recursive function, Axe aborts its sweeping and merging process and uses the lemma to simplify the miter. This may simplify the DAG directly (e.g., if some return value of the loop function can be expressed in closed form), or the lemma may be needed for other rules to fire (e.g., rules that depend on the types of the loop function's return values). Also, the lemma may be helpful in proving merges of nodes supported by the loop function.

[20] It also helps in debugging a failed application of the lemma, since failed attempts to prove `work-hard` hypotheses cause messages to be printed.

Chapter 12

Proofs and Transformations for Two Loop Functions

Using Axe to prove equivalence of programs whose loops are not completely unrolled requires dealing with pairs of loop functions that are equivalent (in the sense that they perform essentially the same computation in essentially the same way). The variables in equivalent loops will be related, usually in such a way that the variables of each loop can be expressed in terms of the variables of the other, meaning that a bisimulation relationship holds. This chapter describes the theorems that Axe proves to connect equivalent loops. The analysis is performed during the sweeping and merging process, when Axe attempts to merge two probably-equal nodes supported by equivalent loop functions.

Before loops can be proved equivalent, it may be necessary to perform loop transformations to synchronize them. For example, if the loops to be compared do not correspond exactly because one loop performs more iterations than the other, or because one loop performs more work per loop iteration, Axe can sometimes apply the loop splitting or constant-factor unrolling transformations, described in Sections 12.2.2 and 12.2.1, respectively. Also, if one side of the miter contains two loop functions (a producer and a consumer) whose composition is equivalent to a single function on the other side of the miter, Axe can sometimes perform a streamifying transformation to combine the producer and consumer into a single loop, which can then be proven equivalent to the loop on the other side of the miter. Section 12.2.3 describes this transformation.

12.1 Proofs Connecting Two Loop Functions

When Axe analyzes two loops together, it first checks that their iteration counts are the same for every test case[1]. (If not, constant factor unrolling or loop splitting may be needed to synchronize the loops.)

If the iteration counts agree, Axe attempts to discover relationships between the variables of the loops that always hold when the loops are executing corresponding iterations. It does this by analyzing traces of the loop functions, as described in Section 13.8. (As always, these traces, even for nested loops, are based on real inputs to the top-level miter.) The result is a set of candidate connections, each of which is a formula over the loop parameters of the two loops (and perhaps their corresponding old variables)[2]. Most commonly, the connections are equalities between parameters of the two loop functions, but other connections also arise. As was the case when generating explanations of loop variables, care is taken to avoid circular chains of equalities.

Given the set of candidate connections, Axe attempts to find the maximal inductive subset. This process is analogous to finding the maximal inductive subset of candidate invariants for a single loop. The proof required for each candidate is to show that it is preserved by the loop bodies of the two loops, assuming that it and

[1] Axe only uses test cases for which both loops are "used."

[2] Axe renames the formal parameters of the loop functions to avoid clashes caused by formals with same names.

all the other connections hold before the loop bodies are executed. The proof also assumes that neither loop exits on the iteration being considered, and it assumes the invariants of the individual loop functions (which were shown to be inductive when the individual functions were analyzed). As is the case when analyzing a single function, claims that fail to prove are removed from the set of candidates, and the process is restarted with the resulting smaller set. This continues until an inductive subset is found.

Axe allows user-supplied connections to be added to the set of candidates before proofs are attempted (as long as they indeed hold on all the test cases). Likewise, the user can specify that certain connections should be removed from the set.

After finding the maximal provable subset of candidates, Axe forms their conjunction. This is the "connection relation" for the two loops, and the fact that it is preserved by the loops bodies constitutes the inductive step of the main proof connecting the two loops.

The next step is to prove that the exit tests of the two loops always agree. (This corresponds to the observation that the loops always seem to have the same iteration count.) For the proof, Axe assumes the connection relation and the invariants for the individual loops. The proof is done using the Axe Prover.

The next step is to generate the set of claims that hold on the final iterations of the two loops (that is, on the values returned by the loops). This is analogous to the situation for a single loop, in which we saw that the loop invariant can often be strengthened for the final iteration, because the exit test is also known to be true. Likewise, the connection between two loops can be strengthened for the final iteration. For example, consider two loops operating on arrays of the same length, where each loop fills in the elements of its array in order, and where the same array data is written by both loops. The connection relation of the loops would specify that the first n elements of the two arrays always agree, where n is the loop index of one of the loops. Initially, n would be zero and the connection would be vacuously true. On each iteration, n would increase, but both loops would write the same data into corresponding array positions, so the connection relation would continue to hold. Now consider what happens on the final iteration, on which both loops exit and for which n will be equal to the length of the arrays. In this case, knowing the value of n allows the connection to be strengthened: when the loops exit, the entire arrays are equal. (The first n elements of an array of length n constitute the entire array.)

As was the case for a single loop, final claims are discovered by the analysis of execution traces (see Section 13.8), and proofs of the final claims are attempted using the Axe Prover. The proof required for each claim is that it follows from the connection relation and the exit tests and invariants of both loops.

Given the proved final claims, it remains to build the main theorem about the functions' return values. The process is analogous to how Axe builds the theorem about a single loop using its loop invariant. Essentially, the conclusion of the theorem says that the connection relation holds on the values returned by the two loops. (Axe deals with the issues of functions returning values as tuples and possibly not returning all of their variables.) The theorem's hypotheses require that the connection hold on the values initially passed in to the two loops, and that each individual loop invariant holds. The proof is by induction, with the inductive step using the fact that the set of candidate connections is inductive.

As is the case for a single loop, the main theorem connecting two loops may mention old variables, and their handling is analogous to what is done in the single loop case.

The theorem relating the two loop functions is used during sweeping and merging, to prove merges of nodes whose supporting nodes call the now-proven-equivalent loop functions. Because the connection lemma may simultaneously characterize several return values of the loop functions, it may be used in the proofs of several different merges.

12.2 Synchronizing Transformations

The proof about two loops requires that they always have the same number of iterations, so that we can talk about relations between their loop variables on "corresponding" iterations. Thus, the proof may need to be preceded by the application of transformations that synchronize the two loops. This section describes such transformations. As always, they apply only to simple loop functions.

12.2.1 Constant Factor Unrolling

It sometimes happens that two loops perform essentially the same computation, but the loop iterations do not directly correspond because one loop does more work per loop iteration. This may occur when one implementation has been optimized by the application of a loop unrolling transformation. This transformation, called "constant factor unrolling," is not to be confused with the "complete unrolling" described in Section 10.4; in particular, constant factor unrolling simply modifies the loop, rather than eliminating it entirely. Constant factor unrolling results in a loop whose body contains several copies of the original loop body; the new loop body thus performs more work. If one loop has been unrolled in this way and is to be proved equivalent to some other loop, it will be necessary to unroll the other loop by the same constant factor.

A similar situation arises when two loops process data in differently sized chunks. For example, while the formal specification of the SHA-1 hash function specifies the processing of data in blocks of 16 words, the corresponding loop in the Java implementation processes one word per iteration, with every 16th word triggering the processing of a complete block. Unrolling of this tight loop, to yield a loop that processes an entire block per iteration, must precede the proof of equivalence.

Axe currently requires the user to specify the unrolling factors (e.g., 16 in the case of the loop from SHA-1), but it automates the rest of the unrolling transformation. In particular, it mechanically generates a new, unrolled version of the loop function and proves it equivalent the the original function by induction. A complication in the process is the possibility that the total number of iterations performed by the loop may not be a multiple of the unrolling factor. In this case, there may be extra loop iterations to deal with. Axe addresses this by including multiple base cases in the unrolled function, one for each possible number of extra loop iterations. Consider the following simple loop function:

```
(defun example (params)
   (if (example-exit params)
       params
       (example (example-update params))))
```

When Axe is asked to unroll this function by a factor of three, it generates:

```
(defun example-unrolled-by-3 (params)
  (if (example-exit params)
      params
      (if (example-exit (example-update params))
          (example-update params)
          (if (example-exit (example-update (example-update params)))
              (example-update (example-update params))
              (example-unrolled-by-3
               (example-update (example-update (example-update params))))))))
```

This function has three bases cases, corresponding to 0, 1, or 2 extra loop iterations; each base case calls `example-update` the appropriate number of times. If 3 or more iterations are to be performed, the new function performs three iterations (by calling `example-update` three times) and then calls itself recursively. When Axe builds the unrolled function it also automatically proves the following theorem:

```
(defthm example-becomes-example-unrolled-by-3
  (equal (example params)
         (example-unrolled-by-3 params))
```

which can be used as a rewrite rule to replace calls of the old function with its unrolled version.

The several bases generated for an unrolled function (and the fact that some of them perform the non-trivial work of calling the update function), make the unrolled function fail to be in simple form. The next step after unrolling is thus to transform the result to simple form.

12.2.2 Loop Splitting

It sometimes happens that the loops in two programs are similar, but one loop performs more iterations. For example, the formal specification of SHA-1 involves padding the input message to make its length a multiple of the block size and then processing each block of the result. In the Java implementation, the full, padded message is never created. Instead, the complete blocks of the message are processed first, and extra bytes are handled separately (by processing them along with the padding data). When proving the equivalence of the two loops, the loop in the specification must be split to synchronize it with the implementation[3]. The split yields a loop that performs enough iterations to process the complete blocks, followed by a second loop that processes the rest of the blocks.

Since each block in SHA-1 is 64 bytes, the number of complete blocks can be found by dividing the input length by 64 and then taking the floor[4]; this value is used as the split amount. Given the split amount, Axe mechanically generates a new

[3] This is after the implementation loop has been unrolled by a factor of 16, as described in the previous section.

[4] In Axe, this would be expressed using 32-bit integer division as `(bvdiv '32 (len input) '64)`.

function that performs the same computation as the original loop but is limited to the given number of iterations. This is analogous to the "limited" function used to force complete unrolling (described in Section 10.4.2). A new parameter is added to count down the number of iterations remaining, and the function exits if that parameter hits 0, or if the loop's exit test is ever true. Axe then proves that calling this function first, followed by calling the original loop function to perform any remaining iterations, is the same as just calling the original function[5]. The split amount used in the transformation (e.g., `(bvdiv '32 (len input) '64)`) need not be a constant.

For SHA-1, the loop that performs the remaining iterations (after the complete blocks are handled) can be completely unrolled, because it executes at most twice[6]. After such unrolling, the operations that deal with padding in the two implementations can be proven equivalent without the need for inductive proofs. Thus, the complicated details of the padding operation are conveniently handled after applying splitting.

12.2.3 Streamifying

It sometimes happens that equivalent computations differ in that one uses two loops where the other uses a single loop to compute the same result. For example, the first loop may produce a sequence of data elements which are consumed in order by the second loop. This whole computation may be equivalent to a single loop that never actually creates the entire sequence in memory but instead produces each individual element just in time for it to be consumed. The single loop can be considered the "streamified" version of the computation. Such a situation arises in the verification of the RC4 stream cipher, and is described in some detail in Section 2.2.2.

[5] Actually, Axe makes an alias of the original function, so that the splitting transformation is not applied again.

[6] Padding the last partial block may cause the message to be extended by an extra block.

The streamifying transformation requires that the producer and consumer satisfy certain properties. For example, the element produced by the producer must not depend on previously generated elements (because, in the streamified computation, those elements are not stored). Also, the consumer must consume the list elements in the order in which they are created; each iteration must use only the `car` of the sequence and must pass the `cdr` of the sequence on to the recursive call. The consumer's exit test can check whether the sequence is empty, but arbitrary tests on the sequence are not allowed, since the streamified function won't build the whole thing.

Using the producer and consumer functions, Axe builds a new function that represents their composition. The new function is usually quite complicated, but it follows the general structure of the consumer, except that it does not take the sequence as an argument. Instead, it has all of the arguments of the producer (except for the sequence). The new function passes around all of the producer arguments, updates them just as the producer did, and generates elements of the sequence on the fly. Where the consumer referenced the `car` of the sequence, the new function uses the value generated from the producer's arguments.

Much of the complexity of the streamified function is for handling the case when the producer stops producing elements before the consumer stops consuming them. If that happens, the new function makes no further changes to the producers' arguments. From that point on, the sequence is considered empty (because all of the produced elements have been consumed), and `nil` is used for all subsequent elements (just as would happen if the original consumer were given a sequence that was too short).

Axe expresses the result of streamification as a rewrite rule that replaces the composition of the producer and consumer with the new function. This rule has been proved in ACL2 for generic producer and consumer functions, but Axe does not yet prove each concrete application of this transformation.

Chapter 13

Test-Based Discovery of Loop Properties

The previous two chapters discussed how Axe proves theorems about loop functions. Those proofs rely on candidate properties, including loop invariants, connection relationships between equivalent loops, and properties that hold on loops' final iterations. This chapter explains how Axe automatically discovers many of those properties.

Unlike static analyses, which analyze code without running it, Axe discovers loop properties dynamically; it generates and analyzes actual execution traces of the given loop functions. These traces correspond to real executions of the entire programs containing the loops. Loop properties discovered in this way are of course not guaranteed to hold on every possible execution; Axe must prove them. Axe can tolerate some failed proofs, but failures increase the time taken by the equivalence checking process. The goal of loop property discovery is thus to produce candidate properties that are true, provable, and helpful in the verification process.

13.1 Dangers: Overfitting and Unrelated Variables

Section 10.3 described how Axe generates traces that represent the executions of loop functions. Each trace records the sequence of values assumed by all of the loop's parameters during the course of a complete execution of the loop on actual data. Taken together, all of the traces for a loop function give information about its typical behavior.

One danger of dynamic analysis is overfitting, in which the generated property fits the trace data too closely. Such a property may fail to cover all possible behaviors and may prevent the generation of a more general correct property. For example, consider the problem of generating numeric bounds for a variable. A naive approach would be to simply find the maximum and minimum value of the variable on any trace and then assert that the variable always lies within that range. But this can be too restrictive. Consider a 16-bit variable, `x`, that represents a word of cryptographic data. It is usually the case that such a data variable can take on any possible value of its type, in this case, any 16-bit word. Thus, the correct numeric bound would simply claim that `x`'s value lies in the interval `[0, 65535]`[1]. However, the traces analyzed by Axe might not include all possible values in that range (since the set of traces is necessarily finite). It may be that the smallest value seen on any test case is, say, 2, and the largest is, say, 65531. In such a case, claiming that `x`'s value lies in the interval `[2, 65531]` would be undesirable, because it would likely be unprovable. To avoid this sort of overfitting, Axe's procedure for generating numeric bounds, described in Section 13.3.2, is more sophisticated. In particular, it avoids treating the set of observed values as a simple, flat set of "samples." Rather, it pays attention to how

[1] Actually, in this case, Axe would use the `unsigned-byte-p` predicate, giving `(unsigned-byte-p '16 x)`.

the values are grouped into traces and the order of the values within each trace.

Another way of addressing overfitting would be to try to prove weaker versions of any claims that fail to prove (perhaps when they are of certain forms, such as claims about bounds). This might still involve wasting time initially, trying to prove claims that are too strong.

Another danger is the generation of properties that, while seemingly true, are not useful, because they connect variables that are not conceptually related. A classic example is the claim that the temperature is always less than the year, which, while true on all test cases, would not be helpful in program verification. Likewise, a claim that some loop index (which is usually small) is always less than some cryptographic word (which is usually represented by a large number) would not be helpful to Axe[2]. Such properties would be generated by a naive approach that considers all possible inequalities between variables and simply reports any that hold on every trace. There may be many such inequalities, and, to avoid the cost of trying and failing to prove all of them, Axe uses a more sophisticated invariant generation approach.

The analyses described in this chapter were implemented in order to support Axe's proofs about cryptographic code. Other analyses are of course possible, and it would not be too difficult to extend Axe to detect more kinds of properties.

13.2 Constant and Unchanged Values

Among the most important properties detected by Axe are loop invariants, each of which is a formula over the formal parameters of the given loop function and their corresponding "old" versions. The next several sections describe how loop invariants are detected from traces.

[2] It would likely be unprovable, because there will usually be some rare inputs that make cryptographic data values contain almost all zeros (and thus represent very small numbers).

Perhaps the simplest possible property of a loop variable is that it is constant, i.e., that it has the same fixed, constant value at every location in every trace. This is quite common, with, for example, the variables that represent the upper or lower bounds of loop indices, or that represent lookup tables (represented as constant arrays). It is also common for the length of a composite object (such as an array) to be constant even when its constituent values are not. When Axe sees a constant value, it always generates a claim to reflect that fact. This seems to be a fairly safe, since the same value was observed in all traces. (Recall that there must be at least two traces). If a loop variable is found to be a constant, it is usually not analyzed further, since the claim that it is that particular constant subsumes all other possible assertions about it (its type, numeric bounds, etc.)[3].

Somewhat similar to constant values are values that are "unchanged per trace." These are values which do not change within a given trace but which may differ between traces (for example, the length of some input variable, which is randomly generated for each test case but which does not change while the program executes). Such values give rise to "unchanged variable" claims expressed using "old variables." For example, if `x` is unchanged within each trace, Axe generates the loop invariant `(equal x oldx)`. Unchanged values are used to push back the other invariants during the invariant improvement process, as explained in section 11.3. Quantities that are unchanged per trace are also important in the detection of certain other properties, as described below.

13.3 Facts About Scalar Values

Values in Axe can be divided into scalars (such as integers) and composite values, such as arrays, sequences, and tuples. Consider first the problem of invariant generation

[3] Of course, the claim that the value is constant may fail to prove, in which case it would be good to have weaker properties to try, but this doesn't happen very often in practice.

for scalar values, which are almost always integers.[4]. Recall that bit-vectors are represented as integer values.

13.3.1 Type Facts

Axe attempts to generate type facts about loop variables. For example, it generally asserts that integer variables are 32-bit bit-vectors (unless this is contradicted by the traces, of course)[5]. If a loop variable never has a negative value (meaning its high bit is never 1, since Java uses the two's complement scheme), Axe further asserts that the value is non-negative. This seems to be a rather safe claim, given that it holds on all of the traces.

13.3.2 Numeric Bounds

As described above, a naive approach to numeric bounds (using the minimum and maximum values observed on any trace) is subject to overfitting. Such an approach simply treats every value in every trace as a independent "sample" and then finds the extrema of the sample set. No attention is paid to which samples occur in which traces, or the order in which samples appear within a trace. Axe's approach to numeric bounds does rely on that information.

Axe first checks whether each trace is the same (that is, whether the sequence of values assumed by the loop variable is the same during every execution of the loop). For example, a loop index which always counts from 1 to 10 would generate the sequence `(1 2 3 4 5 6 7 8 9 10)` for every trace. In such a case, Axe is quite aggressive in generating numeric bounds; it uses the minimum and maximum values

[4] An exception, mentioned in Section 11.7, concerns type specifiers for byte or boolean arrays in Java. These are represented by symbols, such as `'byte`, but such values are almost always constant and thus are easily handled by the methods of the previous section.

[5] Axe could be improved to make better use of type information extracted from the Java class files. This would allow it to deal better with bit-vectors of other sizes, such as Java's 64-bit `longs`.

in the trace. For the example just mentioned, Axe would generate the claim that the loop variable's value lies in the interval `[1, 10]`. The rationale for this is that because every trace shows the exact same behavior, this is likely to be the only possible behavior. (Generating numeric bounds for a loop function in this way would be error prone if there were exactly one trace, but, Axe requires at least two traces.)

If every trace is not the same, Axe is much more conservative. It first checks whether the traces are all either monotonically increasing or all monotonically decreasing[6]. Monotonic traces occur very often for loop indices that count up or down (as is typical in `for` loops). Axe generates claims representing the monotonicity of sequences. For an increasing variable, `x`, the claim is `(not (sbvlt '32 x oldx))`, and for a decreasing `x` it is `(not (sbvlt '32 oldx x))`.

Axe also attempts to generate other upper and lower bounds for the variable. Consider the following three traces for some variable `x`:

```
(0 1 2 3 4 5)
(0 1 2)
(0 1 2 3 4 5 6 7 8 9 10 11)
```

(In the first trace, `x` begins at 0 and is incremented up to 5, at which point the loop exits. The second traces involves fewer loop iterations, and the third traces sees `x` incremented from 0 to 11.) What bounds should Axe generate from these traces? The lower bound is clear: `x` is always greater than or equal to 0, because every trace starts at 0[7].

What about the upper bound of these traces? Clearly there must be some reason that the traces end when they do, and this must depend on the values of other loop

[6] These checks are done using signed 32-bit comparisons, so that negative values (e.g., -1, which is actually represented as 4294967295) are handled appropriately. It would be good to generalize the code to apply to bit-vectors of sizes other than 32 bits.

[7] The lower bound of 0 also follows from the fact that x is monotonically increasing and `oldx` is always 0. Section 13.5 describes Axe's detection of properties of old variables.

variables. To handle this case, Axe forms the sequence of upper bounds (one value per trace):

```
(5)
(2)
(11)
```

and then tries to explain these values using other values that are unchanged per trace. For example, the length of the program's input may take on the following values for the three traces:

```
(21)
(18)
(27)
```

These values are connected to the upper bounds for `x`: in each trace, the upper bound for `x` is 16 less than the input length. Axe would discover this fact. Because such facts are not likely to arise by coincidence, Axe would generate the claim that `x` is less than or equal to `(len input)` minus 16.

13.3.3 Facts about Remainders and Bit Slices

Consider a variable for which every trace is `(0 4 8 12 16)`. Axe will generate the claims that `x` is in the interval `[0,16]` and that `x` is monotonically increasing. However, those facts do not completely characterize the behavior of `x`. It may also be necessary to know that `x` is a multiple of 4. Axe can detect this property. Actually, Axe recognizes the more general case of a sequence in which the values increase by a fixed amount on every step[8]. Values that increase in a stepwise fashion all

[8] The current implementation requires the sequence to be the same for every trace, but that restriction could be perhaps be relaxed.

have the same remainder when divided by the step size (e.g., 0 when dividing by 4). For this example, Axe would generate `(equal (sbvmoddown '32 x '4) '0)`, where `sbvmoddown` is a signed bit-vector remainder (modulo) operator where the division operation rounds toward negative infinity.

Axe also finds cases in which some bit slice, from some bit n down to bit 0, of a variable is always the same constant. Given the three short traces `(126 190)`, `(62)`, `(254 318)`, this would detect the fact that the low 6 bits of all of the values represent the decimal number 62[9].

13.4 Invariants About Composite Values

The values in traces are often composite objects. Recall that each element of a trace is a tuple, with one element for each loop parameter. Each parameter may itself be a composite object: an array, sequence, or even another tuple. Composite objects are especially common when data packing is involved; for example, one parameter of the MD5 hash function's main loop is a list of blocks, each of which is a list of sixteen words, each of which is a 32-bit bit-vector.

Axe's loop property detection handles composite objects. For example, when trying to discover type facts about a composite object, Axe first checks whether its length is the same for every sample. If so, it generates a claim that the length is that particular constant and then descends into the components (since there are always the same number of components) to analyze them recursively[10]. If the lengths of the values are not all the same, Axe generates a very weak claim about the length (that it is a non-negative 32-bit number[11]) and does not examine the components.

[9] Any such fact is also a fact about remainders, but the analysis of bit slices does not require all traces to be the same or all values to be monotonic. The analysis is still cheap, because there are not many possible slices of low numbered bits to check.

[10] For efficiency, Axe avoids examining the individual elements of long sequences of integers.

[11] This reflects the maximum size of a Java array.

13.5 Discovering Claims About Old Variables

The loop invariants dealt with by Axe can mention "old variables," and Axe can discover facts about old variables when it analyzes traces. For example, it can find claims that old variables are constant, are of certain types, or have constant low bits. The analysis uses only the first value of each trace, corresponding to the original value passed into the loop function on that trace.

It is especially important not to generate false claims about the old variables. This is because, while false claims about non-old variables are likely to be dropped because they fail to be inductive, properties of old variables are always trivial to prove inductive (because the old variables do not change in the loop body). Since claims about old variables give rise to hypotheses in the theorem about the loop, it is important to ensure that the claims are always true.

13.6 Explanations

Equalities that explain some loop variables in terms of others are especially helpful to Axe. For example, they allow redundant loop parameters to be dropped, which helps simplify loop functions (as described in Section 11.7). Explanations are also critical to the discovery of loop connections, described in the next section.

The generation of explanations handles composite values. When trying to find an explanation for a variable `x` in terms of other loop variables, Axe first tries to explain the entire value `x`. If that fails, and if `x` is a composite object, Axe tries to find an explanation for the length of `x`. Then, if the length is the same on every test case, Axe tries to find an explanation for each component of `x`. (Ultimately, Axe may be extended to try to explain each individual bit of `x`, if `x` is a bit-vector.)

Likewise, when trying to explain an entire value `z` in terms of some variable `y`, Axe first tries to explain `z` in terms of all of `y`. If that fails and `y` is a composite object, Axe moves on to try to explain `z` using `y's` length or any component of `y`, and so on recursively.

An important step in the process of generating explanations is recognizing that a value to be explained is unchanged per trace. If so, there is little point in trying to explain it using some value that does change within any trace.

It remains to describe how Axe finds explanations between scalar values. To do so, Axe uses a variety of strategies. Consider a variable `x` with the following three traces:

```
(0 1 2 3 4 5)
(0 1 2)
(0 1 2 3 4 5 6 7 8 9 10 11)
```

Assume that some other variable y, has the following traces (for the same three test cases):

```
(5 4 3 2 1 0)
(2 1 0)
(11 10 9 8 7 6 5 4 3 2 1 0)
```

One of the things that Axe does when trying to explain `x` with `y` is to add the values of corresponding traces and check whether the sum is unchanged per trace. Adding corresponding values of `x` and `y` gives:

```
(5 5 5 5 5 5)
(2 2 2)
(11 11 11 11 11 11 11 11 11 11 11 11)
```

It is probably not a coincidence that the sum is unchanged within each trace. In fact, it indicates that `x` and `y` are related, and we are making progress; Axe only needs to find some way of expressing the sum. Since the sum is unchanged per trace, that process uses only values that are unchanged per trace. In this case, Axe finds that the sum is simply the initial value of `y` on each trace, i.e., `oldy`[12]. Thus, `x` can be explained as the difference of `oldy` and `y`.

Axe supports several other kinds of explanations among scalars, including cases where the difference, rather than the sum, of two values is constant per trace, and cases where a value is explainable as a bit slice of another value.

Axe's analysis can find "magic numbers" in explanations. In the example above, the sum was actually equal to `oldy`, but Axe would also detect the case where the sum was `oldy` plus or minus a constant (say, `oldy+107`). This is because Axe's attempt to explain the sum would involve (among other things) subtracting the corresponding values of `oldy` and checking whether the results are always the same constant (e.g., 107). A naive approach to finding such magic numbers (trying all possible constants, or at least all "small" constants) might be quite expensive.

When Axe generates explanations, it takes pains to avoid circularity. That is, it avoids situations where a variable could be explained in terms of itself using the generated equalities. For example, if Axe generates the equality `(equal x (+ 1 y))`, it would avoid also generating `(equal y (+ -1 x))`[13]. Circularity must be avoided if the generated equalities are to be used as assumptions during rewriting. The example just mentioned would cause loops, because `x` would be replaced with `(+ 1 y)`, in which `y` would be replaced with `(+ -1 x)`, giving `(+ 1 (+ -1 x))`. Since `x` appears in `(+ 1 (+ -1 x))`, `x` would be replaced again, and the process would continue forever.

Avoiding circularity is tricky when loop variables are composite objects. For example, if Axe can explain `x` in terms of component 1 of `y`, expressed as `(equal x`

[12] Every old variable is necessarily unchanged per trace.

[13] This example uses `+` for clarity; the real terms would use `bvplus`.

`(nth 1 y))`, it should not later explain `y` in terms of `x` or any component of `x`, such as `(nth 0 x)`, because doing so might cause rewriting loops. (On the other hand, it would be safe to explain `(nth 0 x)` in terms of `(nth 1 y)` while also explaining `(nth 0 y)` in terms of `(nth 1 x)` – because no looping replacements could result.)

To track which explanations are not allowed, Axe maintains the transitive closure of the set of all explanations that have been generated; this allows Axe to quickly check whether a posited explanation will cause circularity.

13.7 Properties That Hold When Loops Exit

As discussed in Section 11.9, the proof about a loop function makes use of so-called "final claims" about the loop, because stronger claims may hold over the loop variables when a loop exits than on arbitrary iterations of the loop. Axe detects candidates for the final claims by examining traces. Of course, to do so, it only examines the last value in each trace. The process consists of searching for explanations; Axe tries to explain the last values of traces in terms of constants and the old variables. For example, if a loop increments its loop index, `x`, exactly ten times, then final value of `x` will be 10 more than the original (old) value of `x`. (The claim clearly holds only on the final iteration of the loop.)

13.8 Discovering Connections Between Two Loops

So far this chapter has discussed how Axe detects properties of single loop functions. It is also critical for Axe to detect relationships between two loop functions that are to be proved equivalent. The process is very similar to how Axe searches for explanations of the variables of a single function. In particular, Axe attempts to explain each parameter of one function in terms of the parameters of the other function, and vice versa, using the explanation detection method previously described. As before, the process handles the lengths and components of composite objects, trying to explain each one and also using them in explanations. Also, as before, the process is careful not to generate circular explanations.

As discussed in Section 12.1, the connection between two loops is often stronger on the final iterations of the loops than on an arbitrary loop iteration. Axe generates candidates for the “final claims” about two loops by examining the last values of the execution traces of the two loops. Again it uses the capability to detect explanations, this time seeking to explain the final values of each loop’s variables in terms of the final values of the other loop’s variables.

The dynamic discovery of connections between two loops (and the ability to prove such properties) enables Axe to perform equivalence checking of programs with non-unrolled loops.

Chapter 14

Conclusion

This dissertation has described Axe, a system for performing equivalence checking of cryptographic programs. Axe is highly automated and has been applied to many real implementations of cryptographic algorithms. The programs verified are widely used and were not written with verification in mind. Axes provides strong correctness results (bit-for-bit equivalence) and can greatly reduce the effort of producing such proofs. It can be used to prove a Java implementation equivalent to an ACL2 formal specification, or to a second Java implementation. With Axe, formal proofs may be a viable alternative to testing for validating cryptographic building blocks.

14.1 Research Contributions

Among the research contributions of Axe are:

- Word-level[1] equivalence checking, which combines rewriting (including the use of several domain-specific simplifications) with techniques from hardware equivalence checking (sweeping and merging) and which operates in phases, working first at the word level before the usual bit-blasting is performed. The use of

[1] Word-level operations deal with multi-bit bit-vectors, e.g., the XOR of two 32-bit values.

a compact DAG representation (common in hardware equivalence checking) allows Axe to apply this method to unrolled cryptographic software.

- Heuristic cutting to prevent the STP tool from timing out when attempting to prove that DAG nodes are equal.
- Inductive equivalence checking (when loops cannot be unrolled), in which Axe performs mechanized induction proofs of loop properties, including connections between equivalent loops.
- Test-based property discovery, including the use of sequences of values from execution traces to detect loop invariants for single loops and connection relationships between corresponding loops.
- The integration of these different verification approaches, allowing hybrid proofs in which some loops are unrolled and others are dealt with inductively. The integrated system also supports the mechanical application of loop transformations to synchronize loops.

The Axe system contains several components that may be of use to others, including the Equivalence Checker and the general-purpose Rewriter and Prover. It also contains a large library of proved simplification rules about common operations (especially bit-vector operations), as well as infrastructure for dealing with JVM bytecode programs (the bytecode parser and decompiler and the M5E model).

14.2 Future Work

Axe provides a common platform for performing equivalence proofs of implementations written in different languages. So far, the only supported languages are Java and ACL2. However, Axe could be extended to handle implementations in other languages by simply implementing a translation layer (perhaps using symbolic execution or decompilation) to extract the DAG representing a program. Since the DAG representation is language-independent, Axe's core reasoning engines could be reused. It would be interesting to apply Axe to implementations in C, C++, and assembly language. Other potential targets include hardware description languages, such as Verilog or VHDL, and the languages of other formal tools.

Another approach to handling other languages would be to compile them to Java bytecode. The Axe system could then perhaps be used as-is. The Scala language [8] might be a good candidate for this, because it is designed to run on the Java Virtual Machine.

Some of Axe's features could perhaps be improved to be more principled and systematic. For example, Axe could support the application of a general class of loop transformations, using something like TVOC's Permute rule (mentioned in Section 3.4). Currently, Axe includes only the transformations needed for the proofs discussed in this thesis. Also, perhaps there is a systematic way to detect which transformations to perform and in what order. This would avoid the manual specification of loop unrolling factors and split amounts (currently needed for the full proofs of MD5 and SHA-1).

Axe could also be extended to discover more kinds of invariants (while still avoiding invariants that would be unlikely to prove). While some loop properties likely require dynamic analysis (because their truth depends on deep facts that are far from obvious statically), Axe could probably benefit from a connection to the most advanced static analysis tools.

A possible improvement to the Axe Equivalence Checker would be to relax the restriction that "probably equal" nodes be "used" on exactly the same set of test cases (as described in Section 8.3.2). Such a change might greatly speed up the full

SHA-1 and MD5 proofs. The challenge would be to still compute the sets of probably equal nodes in an efficient way (avoiding operations that are quadratic in the number of nodes).

Finally, Axe could be applied to other classes of programs. A natural choice might be programs which repeatedly call block ciphers according to “modes of operation.” Pseudo-random number generators might also be amenable to proof using Axe, as might implementations of compression algorithms. Axe is best applied to programs that can be characterized in terms of their input-output behavior. It requires that at least two implementations of the same algorithm exist, and that there be numerous intermediate points of correspondence between the implementations (such as the values computed at round boundaries of block ciphers, which almost always agree between different implementations).

Harder problems worth exploring include equivalence checking of programs with more differences (and more kinds of differences) than the ones described in this dissertation. For example, programs that perform extensive heap manipulations (where perhaps mappings could be found connecting equivalent data structures) could be quite challenging to prove equivalent but might be amenable to analysis using some future heap-aware version of Axe.

Appendix A

Appendix

A.1 Bit-Vector Operators

This section discusses Axe's built-in bit-vector operators. Axe's operators are designed to correspond to the operations performed by real programming languages, such as Java bytecode. They also correspond well to STP's bit-vector operators, which are in turn intended to be compatible with C's operators. Axe's bit-vector library may be of use to other ACL2 users, even those who do not wish to use Axe.

In discussing bit-vectors, we use ACL2's notation for binary numbers, which are preceded by "`#b`". Thus, `#b1110` represents the decimal number `14`. Throughout this dissertation, bit-vectors are displayed with their most significant bit first, and the individual bits within a bit-vector of width n will be numbered from $n-1$ (the leftmost bit) down to 0 (the rightmost bit). Thus, bits 3, 2, and 1 of `#b1110` are all `1`, and bit 0 is `0`.

Axe provides several operators for extracting pieces of bit-vectors. The operator `(loghead n x)` discards everything above the low `n` bits of the bit-vector `x`[1]. For example `(loghead 3 #b0100)` is `#b100`. The operator, `(getbit n x)` extracts the

[1] The name `loghead` is used for historical reasons; `chop` might be a better name.

nth bit of x. For example, (getbit 2 #b0100) is #b1. Finally, (slice high low x) extract the sequence of bits from x that starts at index high and extends down to index low. For example, (slice 2 1 #b0100) is #b10.

Axe's bit-vector operators take parameters that indicate the sizes of the values being operated on. This has several advantages. First, it is always easy to tell the bit width of a term just by looking at it. This is helpful when reading bit-vector expressions, when implementing rewriting strategies about bit-vectors, and when translating goals into the language of the STP theorem prover, which requires all widths and bit indices to be constants. Also, when called on values of incorrect types, Axe's bit-vector operators are able to coerce them to be bit-vectors of the correct sizes. Values that are too big are chopped down, values that are too narrow are implicitly padded with zeros, and non-integral values (such as symbols or lists) are treated as vectors containing only 0 bits. These coercions allow many theorems about bit-vectors, such as the commutativity of bvxor, to actually hold for all values, even values that don't have the correct types. This allows the rewrite rules about bit-vectors to contain fewer type hypotheses, leading to faster rewriting (since relieving hypotheses takes time).

Axe provides a concatenation operator (bvcat highsize highval lowsize lowval), which forms a new bit-vector whose high bits are highval, which should be of size highsize, and whose low bits are lowval, which should be of size lowsize.

Axe provides a "sign-extension" operator, (bvsx newsize oldsize) which takes an argument of size oldsize and extends it to be of size newsize by copying its high bit (its "sign bit") into the new positions. For example, (bvsx 8 4 #b1111) is #b11111111. (The value is extended with ones, since bit 3 of #b1111 is 1.) However, (bvsx 8 4 #b0111) is #b00000111. (The value is extended with zeros, since bit 3 of #b0111 is 0.)

Axe provides three bit shift operators. For left shifting, (shl width x `shift-amount)` shifts `x` to the left by `shift-amount` positions, inserting zeros in the positions opened up by the shift and yielding a result of size `width`. For example, `(shl 8 #b00000011 2)` is `#b00001100`. For right shifting, two functions may be used, depending on whether the positions opened up by the shift should be filled with zeros or with copies of the sign bit of the value being shifted. To fill with zeros, `(shr width x shift-amount)` is used; `(shr 8 #b11000000 2)` is `#b00110000`. To copy the sign bit, `(shr-with-sign width x shift-amount)` is used; `(shr-with-sign 8 #b11000000 2)` is `#b11110000`[2].

Axe provides several operations for bit-vector rotations. The operation `(leftrotate width amt x)` rotates the bits of `x` left by `amt` bit positions, in a field of width `width`. Bits pushed past the left side of the field wrap around to the right. For example, `(leftrotate 8 2 #b11000100)` is `#b00010011`. The `rightrotate` operation is analogous, except the bits are shifted rightward. If the rotation amount of either operation is greater than the specified width, it is first reduced modulo the width. Thus, `(leftrotate 32 37 x)` is the same as `(leftrotate 32 5 x)`[3]. Rotation in a 32-bit field is so common that Axe provides special operators, `leftrotate32` and `rightrotate32`, for that case. As discussed in Section 6.1, Axe has special support for reasoning about code that uses common idioms to perform bit-vector rotations in the absence of true rotation operators.

Axe provides several operators to perform bitwise logical operations on bit-vectors. For example, `(bvand 4 #b0111 #b1010)` is `#b0010`. The operators `bvor`, `bvxor`, and `bvnot` are analogous.

The preceding operators have all dealt with bit-vectors as mere sequences of bits, without regard to what numeric values they might represent. Axe also provides

[2] The Axe operator `shr` corresponds to the Java bytecode operations `IUSHR` and `LUSHR` on `ints` and `longs`, respectively. Likewise, `shr-with-sign` corresponds to the `ISHR` and `LSHR` bytecodes.

[3] When the width is a power of 2, the modulo operation discards all but the low bits of the rotation amount. For example, with a width of 32, only the low 5 bits of the rotation amount are significant. When the width is not a power of 2, all the bits of the rotation amount may be significant.

operators that treat bit-vectors as representing numeric (integer[4]) values.

Axe supports two ways of representing integer values as bit-vectors: the unsigned representation and the signed two's-complement representation. Under the unsigned representation, a bit-vector represents a non-negative integer, with the bit sequence interpreted simply as a binary number. For example, #b1111 would represent the integer 15[5].

Under the signed two's complement representation, both positive and negative integers (and 0) are representable. Using this representation, a bit-vector whose most significant bit is a 0 is taken to represent a non-negative integer; the value is simply the binary number interpretation of all of the bits except the most significant bit. For example, #b0111 would represent the integer 7. A bit-vector whose most significant bit is a 1 represents a negative number. The absolute value of a negative two's complement bit-vector of width n can be found by interpreting it as an unsigned number and subtracting it from 2^n. For example, #b1111 represents the integer -1. It absolute value (namely, 1) can be found by subtracting its unsigned value (namely, 15) from 16 (2^4).

Bit-vectors can of course be compared with equal. Axe also provides eight comparison operators to represent bit-vector inequalities. If bit-vectors are taken to represent unsigned integers, (bvlt size x y) can be used to compare them (bvlt stands for "bit-vector less than"). For example, (bvlt 4 #b0010 #b0011) returns t because 2 is less than 3. The operators bvgt, bvle, and bvge are analogous (they perform the comparisons "greater than," "less than or equal," and "greater than or equal," respectively).

[4] Axe currently has no support for floating-point operations, but such operations are not used in any of the cryptographic algorithms discussed in this thesis.

[5] Indeed, since bit-vectors in Axe are implemented as ACL2 integers, the underlying representation of that bit-vector would actually be the ACL2 number 15.

Axe also provides comparison operators that use the signed representation of bit-vectors. The operation `(sbvlt size x y)` tests whether `x` is less than `y` under the signed representation. For example, `(sbvlt 4 #b1000 #b0000)` returns `t` because -8 is less than 0. The other signed comparisons, `sbvgt`, `sbvle`, and `sbvge` are analogous.

Axe provides several operators that compute mathematical functions of bit-vectors. The operation `(bvplus size x y)` computes the sum of `x` and `y`. For example, `(bvplus 4 #b0010 #b0011)` is `#b0101`, because $2+3=5$. Axe's operators never signal overflow or "saturate" to some maximum value; instead, an overly large result simply "wraps around." For example, `(bvplus 4 #b1000 #b1001)` is `#b0001`, because $8+9=17$, and 17 chopped down to four bits is 1. Essentially, the result is first computed as an integer with unlimited precision and then chopped down to the appropriate bit-vector size. This behavior matches the behavior of the corresponding Java byte code instructions. Rewrite rules dealing with these operations sometimes need to address the problem of overflow.

Axe includes operators for bit-vector subtraction (`bvminus`), arithmetic negation (`bvuminus`, which stands for "bit-vector unary minus"), and multiplication (`bvmult`). If bit-vectors are taken to represent signed numbers in two's-complement form (as in Java), these operations still perform correctly.

Axe includes several operators that compute bit-vector quotients and remainders. If bit-vectors are interpreted as unsigned integers, their quotient can be computed with `(bvdiv size x y)`, which computes the rational number quotient and discards any fractional part. Division by 0 is not defined mathematically, but the ACL2 logic is total, so `bvdiv` must return some value when the divisor is `0`. It turns out that it always returns `0` in such cases, but this behavior should not be relied on.

The operation `(bvmod size x y)` computes the remainder when dividing `x` by `y`. The following formula, which has been proved in ACL2 for all values of `x` and `y`, expresses this fact:

```
(implies (unsigned-byte-p size x)
         (equal (bvplus size (bvmult size y (bvdiv size x y)) (bvmod size x y))
                x))
```

This is the bit-vector analogue of the standard relationship, $y * div(x, y) + mod(x, y) = x$. This relationship happens to hold for `bvdiv` and `bvmod` even when `y` is `0` (In this case, `bvmod` happens to return its first argument[6].)

Axe also provides quotient and remainder operators that use the signed representation. With signed numbers, a choice must be made regarding how to round a negative quotient. One can either discard the fractional part, thus rounding towards `0`, or one can round toward negative infinity. The operators `sbvdiv` and `sbvrem` rounds toward 0, and the operators `sbvdivdown` and `sbvmoddown` round toward negative infinity.

Finally, Axe includes a typed if-then-else operator, `(bvif size test x y)`. If `test` is true[7], `bvif` returns `x`. Otherwise, it returns `y`. Thus, `bvif` is just like `if`, except that it is guaranteed to return a value of width `size`. Having the explicit width saves one from having to consider the widths of all of the leaves of the if-then-else nest (and then taking the maximum), as would be necessary to determine the bit width of a nest of calls to `if`.

A.2 Array Operators

In Axe, a one-dimensional array of bit-vectors is represented as a proper list, but arrays are not manipulated using the standard list operators (`cons`, `car`, `cdr`, etc.) or even `nth` and `update-nth`. Instead, Axe provides special operators for bit-vector arrays:

[6] Actually, if `x` is not already a bit-vector of width `size`, `(bvmod size x 0)` chops `x` down to width `size` and then returns it.

[7] Meaning `t`, not `1`.

```
(bv-array-read element-width len index data)
```

and

```
(bv-array-write element-width len index val data)
```

The `bv-array-read` operator returns the bit-vector of width `element-width` at index `index` from the array `data`, which should have length `len`. The `bv-array-write` operator writes the value `val` at the given index of the given array. As is the case for the bit-vector operators, Axe's array operators have explicit parameters indicating the types of their operands (the length of the array involved and the width of its bit-vector elements). These parameters allow the types of terms to be determined quickly and are helpful in formulating rewriting strategies and when translating Axe's array operations into STP's language. Regardless of its arguments, `bv-array-read` always returns a bit-vector of width `element-width`. Likewise, `bv-array-write` always returns an array of length `len` whose elements are bit-vectors of width `element-width`.

Bibliography

[1] http://code.google.com/p/acl2-books/source/browse/trunk/rtl/.

[2] http://code.google.com/p/acl2-books/source/browse/trunk/coi/super-ihs/.

[3] http://code.google.com/p/acl2-books/source/browse/trunk/misc/expander.lisp.

[4] http://code.google.com/p/acl2-books/source/browse/trunk/misc/prorities.lisp.

[5] Specification for the Advanced Encryption Standard (AES). Federal Information Processing Standards Publication 197, 2001.

[6] AIGER. http://fmv.jku.at/aiger/index.html, 2011.

[7] The Legion of the Bouncy Castle. http://www.bouncycastle.org/, 2011.

[8] The Scala Programming Language. http://www.scala-lang.org, 2011.

[9] Clark W. Barrett, Yi Fang, Benjamin Goldberg, Ying Hu, Amir Pnueli, and Lenore D. Zuck. TVOC: A Translation Validator for Optimizing Compilers. In *CAV'05*, pages 291–295, 2005.

[10] 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*, November 1989.

[11] Al Bessey, Ken Block, Benjamin Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson R. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, pages 66–75, 2010.

[12] Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). `http://www.schneier.com/paper-blowfish-fse.html`.

[13] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007.

[14] Daniel Brand. Verification of large synthesized designs. In *ICCAD*, pages 534–537, 1993.

[15] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*, pages 677–691, 1986.

[16] Computational Logic Inc. Integer Hardware Specification Library. `http://code.google.com/p/acl2-books/source/browse/trunk/ihs/`, 1997.

[17] Galois Connections. Cryptol. `http://www.cryptol.net/`.

[18] Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252.

[19] Patrick Cousot and Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. pages 84–97. ACM Press, 1978.

[20] Jared Davis. Finite Set Theory based on Fully Ordered Lists. In *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '04)*, November 2004.

[21] FIPS PUB 46-3 - Data Encryption Standard. `http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf`.

[22] 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.

[23] Levent Erkök, Magnus Carlsson, and Adam Wick. Hardware/software co-verification of cryptographic algorithms using Cryptol. In *FMCAD'09*, pages 188–191, 2009.

[24] Levent Erkök and John Matthews. Pragmatic equivalence and safety checking in Cryptol. In *PLPV*, pages 73–82, 2009.

[25] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[26] Niels Ferguson, Stefan Lucks, Bruce Schneier, Doug Whiting, Mihir Bellare, Tadayoshi Kohno, Jon Callas, and Jesse Walker. The skein hash function family. `http://www.schneier.com/skein.pdf`, 2010.

[27] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography Engineering*. Wiley Publishing, Inc., 2010.

[28] R. W. Floyd. Assigning meaning to programs. *Communications of The ACM*, 1967.

[29] 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.

[30] C. A. R. Hoare. An axiomatic basis for computer programming. *COMMUNICATIONS OF THE ACM*, 12(10):576–580, 1969.

[31] Alan J. Hu and David L. Dill. Reducing bdd size by exploiting functional dependencies. In *in Proc. DAC*, pages 266–271, 1993.

[32] SRI International. PVS Specification and Verification System. `http://pvs.csl.sri.com/`, 2010.

[33] Guy L. Steele Jr. *Common Lisp. The Language.* Digital Press, second edition, 1990.

[34] Michael Karr. Affine Relationships Among Variables of a Program. *Acta Inf.*, 6:133–151, 1976.

[35] Matt Kaufmann and J Strother Moore. Structured Theory Development for a Mechanized Logic. *Journal of Automated Reasoning*, 26:2001, 1999.

[36] Matt Kaufmann and J Strother Moore. Documentation for ACL2 Version 4.2. `http://www.cs.utexas.edu/users/moore/acl2/v4-2/acl2-doc-major-topics.html`, 2011.

[37] Andreas Kuehlmann and Florian Krohm. Equivalence Checking Using Cuts and Heaps. In *Design Automation Conference*, pages 263–268, 1997.

[38] Andreas Kuehlmann, Senior Member, Viresh Paruthi, Florian Krohm, and Malay K. Ganai. Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification. *IEEE Trans. CAD*, 21:1377–1394, 2002.

[39] Gary Leavens. The Java Modeling Language (JML). `http://www.cs.iastate.edu/~leavens/JML/`.

[40] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Prentice Hall, 1999.

[41] Panagiotis Manolios and J Strother Moore. Partial Functions in ACL2. *Journal of Automated Reasoning*, 31:2003.

[42] Matt Kaufmann and J Strother Moore. ACL2 Version 4.2. `http://www.cs.utexas.edu/users/moore/acl2/v4-2/`, 2011.

[43] John Matthews, J. Strother Moore, Sandip Ray, and Daron Vroon. Verification Condition Generation Via Theorem Proving. In *LPAR'06*, pages 362–376, 2006.

[44] Alan Mishchenko. ABC: A System for Sequential Synthesis and Verification. `http://www.eecs.berkeley.edu/~alanmi/abc/`, 2011.

[45] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *In DAC .06: Proceedings of the 43rd annual conference on Design automation*, pages 532–536. ACM Press, 2006.

[46] Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Improvements to combinational equivalence checking. In *Proc. ICCAD '06*, pages 836–843, 2006.

[47] J Moore. Proving Theorems about Java and the JVM with ACL2. http://www.cs.utexas.edu/users/moore/publications/marktoberdorf-02/index.html.

[48] National Institute of Standards and Technology. Skipjack and KEA Algorithm Specifications. http://csrc.nist.gov/groups/ST/toolkit/documents/skipjack/skipjack.pdf, 1998.

[49] George C. Necula. Translation validation for an optimizing compiler. pages 83–95, 2000.

[50] Niklas Eén, Niklas Sörensson. The MiniSat Page. http://minisat.se/, 2011.

[51] National Institute of Standards and Technology. Cryptographic Algorithm Validation Program. http://csrc.nist.gov/groups/STM/cavp/index.html, 2011.

[52] 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.

[53] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation Validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166, London, UK, 1998. Springer-Verlag.

[54] Hendrik Post and Carsten Sinz. Proving Functional Equivalence of Two AES Implementations Using Bounded Model Checking. In *ICST'09*, pages 31–40, 2009.

[55] R. Rivest. RFC 3121: The MD5 Message-Digest Algorithm. http://www.faqs.org/rfcs/rfc1321.html.

[56] 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`.

[57] Federal Information Processing Standard Publication 180-2. `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf`.

[58] Eric Whitman Smith and David L. Dill. Automatic Formal Verification of Block Cipher Implementations. In *FMCAD '08: Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, pages 1–7, Piscataway, NJ, USA, 2008. IEEE Press.

[59] Diana Toma and Dominique Borrione. SHA Formalization. In *Fourth International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 '03)*, July 2003.

[60] Yichen Xie and Alexander Aiken. Saturn: A sat-based tool for bug detection. In *Computer Aided Verification*, pages 139–143, 2005.

[61] Xiang Yin, John Knight, and Westley Weimer. Exploiting Refactoring in Formal Verification.

[62] Xiang Yin, John C. Knight, Elisabeth A. Strunk, and Westley Weimer. Formal Verification By Specification Extraction. `http://www.cs.virginia.edu/~jck/publications/dsn.2007.ECHO.pdf`.

[63] Qi Zhu, Nathan Kitchen, Andreas Kuehlmann, and Alberto Sangiovanni-vincentelli. SAT sweeping with local observability don't cares. In *DAC*, pages 229–234.