Constructing Semantic Models of Programs with The Software Analysis Workbench

Robert Dockins, Adam Foltzer, Joe Hendrix, Brian Huffman, Dylan McNamee, and Aaron Tomb

Galois, Inc., Portland, Oregon, United States
`{rdockins,acfoltzer,jhendrix,huffman,dylan,atomb}@galois.com`

Abstract. The Software Analysis Workbench (SAW) is a system for translating programs into logical expressions, transforming these expressions, and using external reasoning tools (such as SAT and SMT solvers) to prove properties about them. In the implementation of this translation, SAW combines efficient symbolic execution techniques in a novel way. It has been used most extensively to prove that implementations of cryptographic algorithms are functionally equivalent to reference specifications, but can also be used to identify inputs to programs that will lead to outputs with particular properties, and prove other properties about programs. In this paper, we describe the structure of the SAW system and present experimental results demonstrating the benefits of its implementation techniques.

Keywords: equivalence checking · cryptography · SAT · SMT · symbolic execution · verification

1 Introduction

The Software Analysis Workbench (SAW) is a suite of tools for transforming programs into *formal models* — logical representations of program semantics — and for subsequent analysis of those models. Such models are appropriate for mechanized reasoning about the functional behavior of programs. For example, SAW can be used to answer questions such as the following, with a high degree of automation:

– Is a tricky optimized program equivalent to a trusted reference specification?
– What is an input that will lead to a given location in a program?
– What inputs will yield outputs satisfying a given predicate?
– Did a refactoring cause any semantic change?
– What is an input for which program A, written in language L produces a different output than program B, written in language M?

SAW supports generating models from several source languages, including Java Virtual Machine (JVM) bytecode, Low-Level Virtual Machine (LLVM) bitcode, and Cryptol (a domain-specific language designed for the description of cryptographic algorithms [21]). As a result, SAW can generate models from C and Java programs, along with other languages that target the JVM or LLVM.

The formal models are represented in a dependently-typed functional language, SAWCore, that adopts the same general design as the internal representation used by proof assistants based on type theory.

The philosophy of SAW is to generate generic models of program semantics, independent of a specific analysis task, and then act as a bridge between programs and existing automated reasoning tools, allowing a wide range of transformations along the way. The goal is to be able to perform mostly-automated proofs about subtle code. SAW integrates existing tools with custom implementations of a number of known techniques, along with a variety of novel enhancements to those techniques.

1.1 The Structure of SAW

Translation from programs to formal models in SAWCore takes one of several forms. For programs that are originally written in functional style (such as Cryptol programs), the process is essentially a straightforward compilation into SAWCore. For imperative programs, the current version of SAW depends primarily on symbolic execution with path merging to generate functional terms. When using symbolic execution, the resulting terms are “flat”; all iteration, whether originating from loops or recursive functions, is fully unrolled. If full unrolling is not possible, symbolic execution can simply fail to terminate.

Once programs have been translated to formal models, SAW supports transforming, composing, and evaluating these models in a variety of ways. A built-in rewriter can transform existing terms according to a chosen set of rewrite rules.

Transformation of formal models is generally used to prove properties about programs. The SAW tools have been tuned to proving functional equivalence between programs (and especially to the implementations of cryptographic algorithms), though the tools are well-suited to proving any relationship between the input and output of programs for which the model generation process succeeds.

The rewriting functionality built in to SAW can sometimes be used to complete a proof on its own. For instance, when performing equivalence proofs between similar programs written in different languages, the SAWCore terms generated from those programs are often very similar and require only minor transformations to become identical. Terms in SAWCore are represented as hash-consed Directed Acyclic Graphs (DAGs), so syntactically equivalent terms are immediately apparent, since they are guaranteed to be represented by the same node in the graph of the term. For proofs that cannot be completed with rewriting alone, SAW provides a connection to various automated and semi-automated external tools, including Boolean Satisfiability (SAT) and Satisfiability Modulo Theories (SMT) solvers, to offload proof tasks.

To improve the assurance of rewrite-based proofs, rewrite rules can be proven correct themselves. A rewrite rule can be built from an equality type in the logic, allowing a term of that type to serve as a witness to the validity of the rule. In addition, SAT and SMT solvers can be used to prove the validity of rewrite rules. In cases where assurance requirements are lower, SAW also allows unproven rules.

The translations between source languages, SAWCore, and external theorem provers are illustrated in Fig. 1. The process of constructing models and orchestrating external theorem provers is controlled by a scripting language called SAWScript. It is a straightforward typed functional language with a large collection of built-in functions dedicated to extracting formal models from programs, manipulating those models, and interacting with external theorem provers.

In particular, SAW provides a novel combination of efficient symbolic execution techniques: a shared representation of symbolic program states; eager path merging; deeply-integrated rewriting; and deep integration with efficient bit-level provers based on an And-Inverter Graph (AIG) representation of boolean functions. These features allow it to construct complete semantic models of programs that include heavy use of bit-level operations. The key novelty of SAW is in this combination of techniques, and the primary contributions of our work include:

- Experimental results demonstrating the benefits of DAG term representations, compositional symbolic execution, and the use of a wide variety of back-end proof tools.
- Demonstration of symbolic execution for equivalence checking of cryptographic algorithms across multiple languages.
- A tool made publicly available to the community.

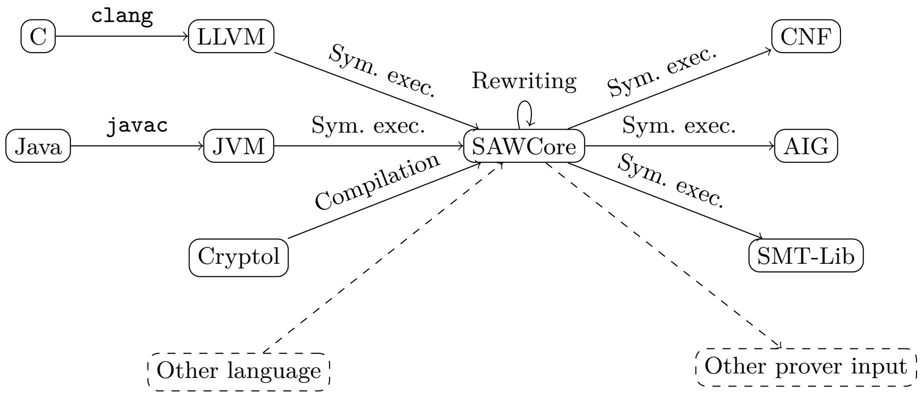

Fig. 1. Architecture of SAW. Source programs are translated to SAWCore, rewritten according to user-defined rules, and sent to external theorem provers. The process is controlled by programs in the SAWScript language.

2 SAWCore

The formal models generated by SAW are represented in an internal modeling language called SAWCore. This language was designed to be an efficient, expressive representation for the semantics of programs originating in a variety of source languages, including languages with sophisticated type systems (such as Cryptol), general recursion (most languages), and complex memory models (such as those appropriate for JVM, LLVM, and most machine languages).

SAWCore is a dependently typed functional language, similar to the core calculi used by languages such as Coq [25], and Lean [22]. It supports user-defined inductive data types, but also has built-in support for a variety of special types such as Booleans, vectors (including extensive support for bit vectors), tuples, and records, for efficient modeling of constructs from software, and for compatibility with external tools that also have special support for these types.

Although SAWCore is intended primarily as an internal representation for program semantics, it also has a concrete syntax with some conveniences to simplify development of libraries of common operations and rewrite rules.

The ability to use high levels of abstraction in SAWCore makes models more compact and easier to transform according to the properties of those abstractions. Ultimately, however, many of the program analyses performed by SAW can largely be represented in first-order form (and the external provers we use generally operate on first-order formulas), so SAW includes back ends for translation of a subset of SAWCore into first-order representations such as the And-Inverter Graph External Representation (AIGER) and SMT-Lib formats. Broadly speaking, the translatable subset consists of functions with domains and ranges that are made entirely of finite bit vectors (potentially aggregated into higher-level vectors, tuples, or records).

We are also considering adding the ability to export terms to Lean or Coq for interactive proof. These will stay at a high level of abstraction and make use of higher-order features, motivating a delay in lowering terms to first-order form.

3 Symbolic Execution

SAW makes heavy use of symbolic execution to translate imperative programs to SAWCore. Unlike most implementations of symbolic execution, we ultimately generate a single model of the symbolic state of a program, for all paths explored, rather than generating a separate symbolic state for each path. This has the advantage of capturing the entire semantics of a program, which is useful for functional correctness verification. However, it has the disadvantage of leading to more complex symbolic states, making it potentially less effective as a bug-finding tool for identifying potential assertion violations on specific paths. Therefore, SAW tends to be effective for programs that can be exhaustively explored by symbolic execution, but less scalable for programs that cannot. Our approach is similar to that used by bounded model checkers in this respect, though the bounds are all provided by the program rather than the model checker. Termination is determined by the satisfiability of loop conditions as each loop iteration executes, so symbolic execution will fail to terminate if the underlying program does not terminate, or if it has a sufficiently complex termination condition.

3.1 Shared Terms

The implementation of SAWCore uses a DAG structure to represent terms, and uses memoization to guarantee that identical subterms are represented with a single graph node. In addition, the SAW system uses a single term node database for all terms generated within a session. Therefore, when comparing two similar programs, semantically identical portions of those programs are immediately identified (even when the original programs are written in different languages). As detailed in Sec. 7.2, this representation is critical for symbolic execution of most cryptographic code, to avoid exponential blow-up. For non-cryptographic code without extensive iteration, the DAG representation is less critical, but still helpful for reducing model sizes.

3.2 Postdominator-Based Merging

To support generating a single model of program semantics, rather than a model of the symbolic state of each individual path, the symbolic execution infrastructure in SAW uses path merging at every node in the control-flow graph that immediately post-dominates more than one other node. To facilitate path merging, we translate the original program into a modified representation which includes special symbolic execution instructions in place of the original branch instructions. Each branch includes a merge point as well as an initial target, and each path that executes starting from that branch instruction will pause when it reaches the merge point. When all paths leading to a single merge target instruction have completed, the simulator merges their symbolic states. The result is a single logical formula describing the final state of the program in terms of its initial state (with free variables denoting arbitrary initial values).

3.3 Memory Models

One of the characteristics that tends to distinguish systems based on symbolic execution from those based on, for instance, weakest preconditions or strongest postconditions (the latter of which is roughly equivalent to symbolic execution), is the use an *implicit* instead of an *explicit* memory model. In an implicit memory model, the mapping between names and (potentially symbolic) values is tracked directly by the symbolic execution system, perhaps as a map data structure in the host language. Imperative updates, then, can be destructive updates to this map instead of existentially-quantified equalities in the generated verification conditions (as would be the case with strongest postconditions).

The ability to destructively update the internal simulator state can make symbolic execution more efficient than strongest postcondition calculation for typical imperative programs, and allows symbolic state expression to remain quantifier-free. However, if imperative updates occur to symbolic addresses, the size of verification conditions can explode with explicit case splitting expressions. Therefore, the ability to trade off between implicit and explicit memory models provides a flexibility advantage.

SAW currently has several implicit memory models, and does not implement an explicit memory model. As future work, we are considering implementing built-in data types in SAWCore that present similar functionality to the current implicit memory models but in a way that would allow them to be directly embedded into SAWCore terms, and therefore be used as part of an explicit memory model. Such terms might be difficult for SMT solvers, but could allow interactive provers to tackle more complex programs, rather than ruling out such cases entirely.

3.4 Path Feasibility Checking

Like many symbolic execution systems, SAW supports (optional) path feasibility checking. The process of executing a conditional branch instruction involves adding the relevant branch condition to the accumulated path condition of each path, and can include checking that condition for satisfiability.

For complex but satisfiable path conditions, full satisfiability checking can be expensive, so SAW also supports another option: translating the path condition term to AIG format and checking for syntactic equality with `False`. The translation to AIG form necessarily includes common simplifications such as constant folding and beta reduction as well as simple representations of bit-level operations such as shifting and masking. In code that uses bit-level manipulation heavily, this operation can frequently suffice to determine path feasibility. For example, consider a program that iteratively performs a logical right shift on a condition. For a bit vector of a fixed size, this operation will yield zero after a fixed number of iterations, making the condition `False` (for a C or LLVM program), and equivalence to `False` is often immediately apparent in AIG format.

3.5 Example

As a simple example of a task that SAW is ideally suited for, consider the POSIX Find First Set (FFS) function, for finding the index of the first bit set in a word. This function is implemented in many standard C library variants. Some implementations iterate over the bits of the word, such as shown in the left column of Fig. 2.

Other implementations avoid loops by masking off bits of the word in chunks, such as shown in the right column of Fig. 2. These two functions compute the same result using dramatically different techniques.

SAW can translate each of these programs to a formal model using symbolic execution and prove the models equivalent with a SAT solver in a fraction of a second (using the short script that appears in Sec. 5).

```c
int ffs_ref(int w) {
  int cnt, i = 0;
  if(!w) { return 0; }
  for(cnt = 0; cnt < 32; cnt++) {
    if(((1 << i++) & w) != 0) {
      return i;
    }
  }
  return 0;
}
```

```c
int ffs_imp(int w) {
  char n = 1;
  if (!(w & 0xffff)) {
    n += 16; w >>= 16;
  }
  if (!(w & 0x00ff)) {
    n += 8;  w >>= 8;
  }
  if (!(w & 0x000f)) {
    n += 4;  w >>= 4;
  }
  if (!(w & 0x0003)) {
    n += 2;  w >>= 2;
  }
  return (w) ? (n+((w+1) & 0x01)) : 0;
}
```

Fig. 2. Reference and efficient implementations of the FFS algorithm which compute the same result using different techniques.

4 Compositional Symbolic Execution

For functional languages, symbolic execution is naturally compositional. Symbolic execution essentially amounts to a non-standard reduction strategy, and any application expression with a name on the left-hand side can be either inlined or treated as uninterpreted.

For imperative languages, the problem is trickier. We would like to treat the target of a function call abstractly, referring to it with an uninterpreted function symbol in our logic. It would also be convenient to provide only some facts about that function, rather than a complete definition. However, doing this automatically is tricky in an imperative setting with an implicit memory model: although treating an imperative program as a pure function can often be straightforward when its inputs and outputs are known, it can in general be undecidable to determine those inputs and outputs automatically.

Therefore, our strategy to compositional verification is to allow users to provide descriptions of the inputs and outputs of a procedure in the imperative language, and then describe the logical function that transforms those inputs to outputs. This function can be an arbitrary expression, including, if desired, uninterpreted function symbols.

Given such a description, we can do two things. We can symbolically execute the procedure being described, given arbitrary contents of the inputs, to derive a term denoting the symbolic values of the outputs. We can then (attempt to) prove that this resulting term satisfies any property we desire.

Alternatively, we can use the same description of a procedure during the symbolic execution of one of its callers. When the symbolic execution engine encounters a function call, it can simply apply the provided expression to the appropriate (symbolic) values of the state elements that form its inputs, and store the resulting term in the portion of the simulator state corresponding to its outputs. Thus, the symbolic execution engine can process procedure calls without examining the callee, and may (as one option) simply use an uninterpreted function to describe the semantics of the callee.

A more general approach is also possible: the semantics of a procedure can be represented by a function that takes in all of that procedure's arguments plus the current heap and returns the procedure's return value plus a new heap. This allows automated composition, but trades off the efficiency possible with an implicit representation of the heap. For programs that use linked data structures or unbounded memory allocation, however, this approach would be effective in cases where the current one is not. We plan to explore this approach more in future work.

5 SAWScript

The process of model generation and transformation in SAW, and the interaction with third-party proof tools, is coordinated by a scripting language called SAWScript. The language is a simply-typed functional language, with an interpreter that can be used either in batch mode or through an interactive Read-Evaluate-Print Loop (REPL).

Many of the built-in functions in SAWScript have externally-visible effects, and these effectful commands are combined with a monad-like construct. Unlike other languages with this approach to combining effectful computations, SAWScript has no facility for user-defined monads, a decision we made to reduce cognitive load. To the SAWScript user, the types of effectful commands simply restrict their use to specific contexts.

One central built-in type in SAWScript is `Term`, representing a SAWCore term. Most built-in functions produce, modify, or consume `Term` values. From the SAWScript point of view, `Term` is a single type, but each `Term` also has a SAWCore type internally. Each `Term` is type-checked according to the SAWCore type system as it is constructed, but the internal type of a `Term` can change without the underlying SAWScript program changing if the structure of the program under analysis changes.

SAWScript also has a tight connection with Cryptol, a language originally developed for the high-level description of cryptographic algorithms, but which is also very convenient for description of any algorithm that operates on fixed-size bit vectors. Cryptol syntax provides a convenient way to construct SAWCore terms that provides type inference and has less syntactic overhead than SAWCore. Existing SAWCore terms can be used in subsequent Cryptol expressions, allowing Cryptol to be used as convenient "glue" around SAWCore terms extracted automatically from programs.

A variety of commands exist for extracting `Term` objects from imperative programs. The simplest work only on a limited set of programs but are completely automatic, translating an imperative function to a lambda abstraction with a type isomorphic to that of the original program. An alternative interface allows more control over symbolic execution, allowing the user to place either symbolic or concrete values into the program state, symbolically execute a function, and then read out components of the final state. The third interface allows for compositional reasoning following the approach described in Sec. 4.

Given a `Term`, SAWScript provides commands to perform rewriting with a given set of rules, unfold abstract named subterms, perform beta reduction, or export it in various external formats. Proving the validity of `Term` values is a central activity in SAW, and a `ProofScript` monad provides a mechanism for chaining simple tactics together to complete a proof. The final tactic in a `ProofScript` can be `trivial` to indicate that the preceding tactics should have reduced the term to `True`, or a tactic that invokes an external prover on the residual term. If a proof fails (or if a `Term` is satisfiable when using the `sat` command), counter-examples are presented in terms of variables from the original program.

Fig. 3 shows a short script that compares the FFS implementations from Sec. 3.5 for equivalence. The `llvm_extract` command translates a simple LLVM function into SAWCore, and the `abc` primitive is a `ProofScript` value that instructs the system to perform the proof automatically using ABC. Expressions between double curly braces are in Cryptol syntax and automatically translated to SAWCore terms.

```
m <- llvm_load_module "ffs.bc";
ref <- llvm_extract m "ffs_ref" llvm_pure;
imp <- llvm_extract m "ffs_imp" llvm_pure;
let thm = {{ \x -> ref x == imp x }};
time (prove_print abc thm);
```

Fig. 3. SAWScript code to compare FFS implementations.

6 Implementation

The SAW implementation brings together a symbolic execution system for the JVM; a similar system for LLVM; an implementation of the SAWCore language, including a rewriting engine; an interpreter for the SAWScript language; and an interpreter for the Cryptol language. All of these components are written in Haskell, and total around 70k Lines of Code (LOC). In addition to this Haskell code, SAW builds heavily on (and statically links with) the ABC system [5], which consists of around 480k LOC in C. ABC is used in particular to represent AIG data structures.

The current implementation can export SAWCore models to ABC, other tools supporting the AIGER format, SAT solvers supporting the DIMACS Conjunctive Normal Form (CNF) format, model checkers that use sequential AIG models, and SMT solvers that use the SMT-Lib2 format (including invocation support for ABC, Boolector, CVC4, MathSAT, Yices, and Z3).

The entire SAW system is publicly available under the 3-clause BSD license:

- An overview and tutorial: `http://saw.galois.com`
- Complete source code: `http://github.com/GaloisInc/saw-script`

6.1 Current Limitations

Because the development of SAW has been driven by the goal of automatically proving properties about cryptographic algorithms, the scope of programs it can effectively model is currently restricted in several ways.

Symbolic Termination. Because symbolic execution is the key technique in SAW for translating imperative programs into formal models, we can successfully generate formal models only in cases where symbolic execution terminates. Symbolic execution is guaranteed to terminate when control flow does not depend on symbolic values (and when the program terminates for concrete values), but it may fail to terminate in other cases.

Memory Layout. The formal models generated by SAW must currently work over data composed of fixed-size bit vectors (potentially aggregated into larger data structures). Because of this, programs that operate over linked data structures such as lists or trees can only be analyzed for specific, fixed layouts.

Exceptions. Exceptions in both JVM and LLVM are largely unsupported. Code under analysis can throw exceptions, as a way of indicating invalid paths, but the symbolic execution engines do not track or invoke exception handlers.

Floating Point. The floating point instructions in JVM and LLVM are supported only for concrete values.

7 Experiments

SAW provides a novel combination of symbolic execution techniques. Although each has been at least proposed in prior work, the performance of each on concrete benchmarks is less well-understood. In this section, we describe how SAW performs on several benchmarks that show the benefits of its design choices. In all experiments, we set a time limit of 1500s, and indicate times longer than this limit with “T/O”.

7.1 Experimental Subjects

We have focused on using SAW for proofs about cryptographic algorithms and implementations, so the chosen benchmarks come from that domain.

FFS Two C implementations of the FFS algorithm (shown earlier), taken from standard C library implementations, compared for equivalence. The implementations are each a single function, so the proof of equivalence uses a monolithic strategy.

AES In-house implementations of the Advanced Encryption Standard (AES) block cipher in C and Cryptol, compared for equivalence using a monolithic proof strategy.

SHA-384 Three implementations of the SHA-384 hash function in C (from the `libgcrypt` library), Java (from Bouncy Castle), and Cryptol, compared for equivalence using both monolithic and compositional proof strategies. This proof covers just the inner loop of the compression function.

ZUC Implementations of two versions of the ZUC stream cipher, in C (from the official reference implementation) and Cryptol, compared for equivalence using both monolithic and compositional proof strategies. Version 1.4 of the algorithm had a bug related to non-injectivity of the key expansion function. We show a proof of the injectivity of the key expansion routine in version 1.5, and an example of non-injectivity in version 1.4 (by automatically producing two inputs for which the key expansion function returns the same output).

ECDSA In-house implementations of the Elliptic Curve Digital Signature Algorithm (ECDSA) over the NIST P-384 curve, written in Cryptol and Java, compared for equivalence using a compositional strategy. We also compare a subset of this implementation for equivalence using both a monolithic and a compositional strategy.

All of the in-house implementations are in the `examples` directory of the `saw-script` repository on GitHub cited in Sec. 6.

7.2 Shared Term Representation

Representing programs using shared (DAG) terms is one of the critical features of SAW. Because symbolic execution unrolls loops, many similar or identical subterms appear in the final program model. Table 1 shows how the shared and unshared sizes of the verification conditions for the following proofs compare. We also show the overall number of code lines, script lines, and total execution time required for each proof. In some cases, such as the SHA-384 equivalence proof, the improvement due to shared terms is simply a significant performance benefit; in other cases, such as the ECDSA equivalence proof, it makes proofs feasible that would otherwise be intractable.

Note that, although most of these benchmarks are equivalence proofs, two are not. The ZUC 1.4 example finds a specific input for which the key expansion function is not injective, and the ZUC 1.5 example shows that the improved key expansion function has been made injective. Both of these cases work directly on the C code without making use of a separate specification (other than a one-line statement of the injectivity property of the key expansion function).

	Term Size		Lines		
Benchmark	**Shared**	**Unshared**	**Code**	**Script**	**Proof Time**
FFS equivalence	6.48×10^2	9.90×10^3	18	5	0.012s
ZUC equivalence	3.96×10^4	3.55×10^6	620	152	6.443s
ZUC 1.4 bug	1.83×10^4	1.27×10^7	263	79	1.692s
ZUC 1.5 injectivity	1.83×10^4	1.30×10^7	263	78	7.047s
AES equivalence	6.67×10^5	2.09×10^{38}	1301	55	901.923s
SHA-384 equivalence	3.39×10^4	6.64×10^5	979	309	8.619s
ECDSA equivalence	3.03×10^5	2.76×10^{273}	4305	1526	311.009s

Table 1. Shared and unshared term sizes, execution times, and script sizes for benchmarks. Proof times are for our original proof scripts, each of which may use several different provers.

7.3 Compositional Proofs

We show the effects of compositional reasoning on several examples on Table 2. Compositional reasoning can split one large proof into several smaller proofs. Because each proof must be processed separately, compositional reasoning can make small equivalence proofs slower to complete (as in the SHA-384 example). However, some proofs that are feasible monolithically are faster when done compositionally (such as ZUC using Z3 and the ECDSA subset using ABC). Most importantly, larger proofs tend to be intractable when done monolithically (such as the ECDSA proof using Z3) but become tractable using compositional reasoning. The full ECDSA proof mentioned in the previous section is compositional. A monolithic attempt at the same proof runs out of memory before even generating a theorem to prove, much less invoking a solver to discharge it.

		Proof Time	
Benchmark	**Prover**	**Compositional**	**Monolithic**
ZUC equivalence	ABC	13.873s	13.064s
ZUC equivalence	Z3	4.445s	5.371s
SHA-384 equivalence	ABC	T/O	T/O
SHA-384 equivalence	Z3	25.219s	7.750s
ECDSA equivalence (subset)	ABC	27.220s	105.284s
ECDSA equivalence (subset)	Z3	75.385s	T/O

Table 2. Execution time for compositional and monolithic equivalence checking. These benchmarks use a single prover, rather than an optimized set, so the proof times differ from those in Table 1.

7.4 Prover Comparison

We have included support for multiple provers within SAW because each tends to be efficient on a different class of applications. In particular, the purely propositional ABC tends to be the most efficient for small cryptographic primitives that primarily perform bit-level operations, whereas SMT solvers become more efficient for larger programs in which compositional verification is necessary. Table 3 shows the time taken by each of five provers on several benchmarks. For the AES benchmark, we used the equivalence checking interface to ABC to compare two distinct circuits. For all other benchmarks, we generated a single formula for the property to be checked.

Benchmark	**ABC**	**CVC4**	**Yices**	**Z3**	**Picosat**
FFS equivalence	0.013s	0.015s	0.018s	0.021s	0.035s
ZUC equivalence	14.451s	T/O	4.196s	6.952s	11.311s
ZUC 1.4 bug	1.692s	3.073s	0.599s	1.991s	2.961s
ZUC 1.5 injectivity	7.047s	T/O	2.051s	1.679s	4.629s
AES equivalence	901.923s	T/O	T/O	T/O	T/O
SHA-384 equivalence	T/O	T/O	56.368s	8.813s	T/O
ECDSA equivalence (subset)	26.743s	T/O	42.925s	74.591s	35.470s

Table 3. Relative prover efficiency. These are the same benchmarks as in Table 1, but with a single prover instead of potentially several. All proofs are monolithic, since ABC and Picosat do not support uninterpreted functions.

8 Related Work

Symbolic execution has been used since at least the 1970s as a technique for software analysis [17]. Many systems have used symbolic execution to prove properties about programs, detect bugs, and guide test generation. Of these, the KLEE tool [7] for LLVM is a representative example, and one of the most robust. Unlike SAW, KLEE focuses on checking specific properties of individual execution paths rather than generating complete models of programs that can be used for a variety of purposes. Others have investigated state merging in symbolic execution [15,19], but have used it more to improve the efficiency of path-based analysis based on symbolic execution, rather than for generation of complete semantic models.

The approach taken by bounded model checkers, such as CBMC [10] and LLBMC [13], is in some ways more similar to that of SAW. Bounded model checkers also tend to construct models of program semantics that are complete up to a certain bound. Model checkers frequently focus on temporal properties of concurrent systems, at the expense of efficiency when reasoning about complex, non-concurrent systems. SAW supports precise, efficient reasoning about sequential code but does not support concurrency or temporal properties.

Contract-based software verification tools such as Frama-C [18], VCC [11], the Java Modeling Language (JML) tools [6], and KeY [1] are very flexible with respect to the sorts of properties they can prove. They typically require significantly more manual effort than SAW for problems supported by both approaches (in the form of manual annotations or user-assisted proofs), but can handle a wider range of properties.

Unlike all of the verifiers mentioned so far, the goal of SAW is to construct a full model of the underlying program, separate from any specific verification task, and then perform any desired analysis on that model. However, some other tools have taken a similar approach to SAW. The most similar is Axe [24], which also aims at comparing cryptographic algorithms for equivalence. Axe uses ACL2 instead of type theory as its internal logic, and is not publicly available. Myreen et al. [23] and Hardin et al. [16] have both described decompilation of low-level imperative languages into logic, using techniques similar to those of SAW, though neither has a similar degree of integration. In the narrower domain of equivalence checking, LLVM-MD [26] used somewhat similar techniques for LLVM translation validation.

Both Why3 [14] and Boogie [20] have very similar goals to SAWCore. They are aimed at modeling the semantics of various source languages and providing easy connection to existing theorem provers. However, both languages use imperative constructs in the modeling language to encode the imperative constructs from the source language. Some standard program analysis techniques (e. g., dataflow analysis) are easier to implement on an imperative language, but verification must generally be annotation-based, and use, for instance, an approach like the weakest precondition calculus. Therefore, imperative modeling languages are not well-suited to the rewrite-based philosophy that we embrace. SAWCore tends to be well-suited to different classes of programs than Why3 or Boogie. Implementing a translator from either Why3 or Boogie to SAWCore could allow for the best of both worlds.

In the specific application domain of cryptography, several proof tools exist. One example is EasyCrypt [3], which allows high-level reasoning about cryptographic algorithms in the abstract, but does not allow proofs about existing concrete implementations.

In the realm of implementation verification, Appel recently proved the SHA-256 code in OpenSSL [2] equivalent to a high-level specification using the Verified Software Toolchain (VST), which provides a strong reasoning path between high-level cryptographic notions and concrete implementations written in C, depending on only the relatively small trusted code base (TCB) of the Coq theorem prover. The TCB of SAW is much larger. However, using VST to show equivalence between the abstract definition of SHA-256 and the C implementation required around 6,500 lines of manual proof, whereas equivalence proofs in SAW tend to be mostly automated. Our hope is that, in the long run, it will be possible to achieve a better balance between TCB size and automation, realizing the best of both worlds.

Relatedly, the miTLS project has created an implementation of Transport Layer Security (TLS) verified to be equivalent to a high-level specification [4]. The proof concerns a custom implementation written in F* and the tools could not be used to verify existing implementations in other languages.

The implementation of SAW described in this paper grew out of previous work on verifying Cryptol programs [12] and is the second iteration of a system briefly outlined in a previous extended abstract [8].

9 Conclusions and Future Work

We have shown that SAW can perform efficient equivalence checking and bug finding on a variety of real-world examples written in several programming languages. It achieves this by combining a collection of known but not previously integrated symbolic execution and program modeling techniques and connecting to a wide range of state-of-the art theorem provers.

Currently, however, SAW is most applicable to a restricted class of programs: those with finite, fixed input and output types that terminate under symbolic execution. Our primary intended direction of future work is to relax the restrictions. To ease the restriction on termination under symbolic execution, we plan to translate at least some iterative programs into explicit uses of fixpoint operations in the logic. This will allow us to generate models of more programs at the expense of more difficult reasoning about the resulting models. To allow general recursion in the context of our logic, while still allowing the logic to be used for proofs, we are considering adopting the approach of Zombie [9].

To ease the restriction on finite, fixed input and output types, we plan to extend SAW with the ability to generate formal models that include the heap as an explicit parameter and result. In conjunction with explicit fixpoint operations, this change should allow SAW to generate models of essentially arbitrary programs. It will, however, place a higher burden on the proof infrastructure required to do analysis of those models. Inductive proofs and complex reasoning about arrays will become much more important. Therefore, proofs about such models may only be feasible with interactive or semi-interactive proof tools, and we plan to explore emitting SAWCore models in the language of a proof assistant such as Coq or Lean.

Acknowledgments Much of the work on SAW and Cryptol has been funded by, and design input was provided by the team at the NSA's Trusted Systems Research Group, including Brad Martin, Frank Taylor, Sean Weaver, and Jared Ziegler.

References

1. Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Herda, M., Klebanov, V., Mostowski, W., Scheben, C., Schmitt,

P.H., Ulbrich, M.: The KeY platform for verification and analysis of Java programs. In: Verified Software: Theories, Tools, and Experiments (VSTTE 2014). pp. 1–17. No. 8471 in Lecture Notes in Computer Science, Springer-Verlag (2014)

2. Appel, A.W.: Verification of a cryptographic primitive: SHA-256. ACM Transactions on Programming Languages and Systems 37(2), 7:1–7:31 (Apr 2015)
3. Barthe, G., Grégoire, B., Heraud, S., Béguelin, S.Z.: Computer-aided security proofs for the working cryptographer. In: Proceedings of the 31st Annual Cryptology Conference (CRYPTO 2011). pp. 71–90. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
4. Bhargavan, K., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y.: Implementing tls with verified cryptographic security. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP). pp. 445–459 (May 2013)
5. Brayton, R.K., Mishchenko, A.: ABC: An academic industrial-strength verification tool. In: Proceedings of the 22nd International Conference Computer Aided Verification (CAV 2010). Lecture Notes in Computer Science, vol. 6174, pp. 24–40. Springer (2010)
6. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer 7(3), 212–232 (2005)
7. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008). pp. 209–224. USENIX Association, Berkeley, CA, USA (2008)
8. Carter, K., Foltzer, A., Hendrix, J., Huffman, B., Tomb, A.: SAW: the software analysis workbench. In: Proceedings of the 2013 ACM SIGAda annual conference on High Integrity Language Technology (HILT 2013). pp. 15–18 (2013)
9. Casinghino, C., Sjöberg, V., Weirich, S.: Combining proofs and programs in a dependently typed language. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2014). pp. 33–45 (2014)
10. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004). Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004)
11. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Theorem Proving in Higher Order Logics, 22nd International Conference (TPHOLs 2009). Lecture Notes in Computer Science, vol. 5674, pp. 23–42. Springer (2009)
12. Erkök, L., Matthews, J.: Pragmatic equivalence and safety checking in Cryptol. In: Proceedings of the 3rd Workshop on Programming Languages Meets Program Verification (PLPV 2009). pp. 73–82 (2009)
13. Falke, S., Merz, F., Sinz, C.: The bounded model checker LLBMC. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering, (ASE 2013). pp. 706–709. IEEE (2013)
14. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: Proceedings of the 22nd European Symposium on Programming (ESOP 2013). Lecture Notes in Computer Science, vol. 7792, pp. 125–128. Springer (Mar 2013)
15. Hansen, T., Schachte, P., Søndergaard, H.: State joining and splitting for the symbolic execution of binaries. In: 9th International Workshop on Runtime Verification (RV 2009), pp. 76–92. Springer Berlin Heidelberg (2009)

16. Hardin, D.S.: Reasoning about LLVM code using Codewalker. In: Proceedings of the 13th International Workshop on the ACL2 Theorem Prover and Its Applications. Electronic Proceedings in Theoretical Computer Science, vol. 192, pp. 79–92 (Oct 2015)
17. King, J.C.: Symbolic execution and program testing. Communications of the ACM 19(7), 385–394 (Jul 1976)
18. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing 27(3), 573–609 (2015)
19. Kuznetsov, V., Kinder, J., Bucur, S., Candea, G.: Efficient state merging in symbolic execution. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2012). pp. 193–204 (2012)
20. Leino, K.R.M.: This is Boogie 2. Tech. rep., Microsoft Research (June 2008)
21. Lewis, J., Martin, B.: Cryptol: high assurance, retargetable crypto development and validation. In: Proceedings of the IEEE Military Communications Conference (MILCOM 2003). vol. 2, pp. 820–825 Vol.2 (Oct 2003)
22. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover. In: Proceedings of the 25th International Conference on Automated Deduction (CADE-25). Berlin, Germany (2015)
23. Myreen, M.O., Gordon, M.J.C., Slind, K.: Decompilation into logic - improved. In: Proceedings of the 12th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2012). pp. 78–81. IEEE (2012)
24. Smith, E.W.: Axe: An Automated Formal Equivalence Checking Tool for Programs. Ph.D. thesis, Stanford University (2011)
25. The Coq development team: The Coq Proof assistant reference manual. LogiCal Project (2004), `http://coq.inria.fr`, version 8.0
26. Tristan, J.B., Govereau, P., Morrisett, G.: Evaluating value-graph translation validation for LLVM. In: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2011). pp. 295–305 (2011)