Towards Automated Reasoning About Machine Code Using SMTCoq

1st Humam Alhusaini
University of Texas at Dallas
Humam.Alhusaini@UTDallas.edu

***Abstract*—Manually providing formal guarantees for machine code through a formalized ISA is often time-consuming and difficult. Despite the fact that SMT solvers are able to solve large, complex theorems concerning bit-vectors, the expressions in goals typical of machine code analysis are often too complex to directly translate to SMT-LIB, the primary format SMT solvers analyze. This suggests that preprocessing goals to a more primitive format would allow analysis by SMT solvers, thus allowing the automation of formal analysis for machine code. This paper presents an in-progress preprocessing tactic for the Picinæ framework that canonicalizes its abstractions to SMTCoq's native bit-vector type. By creating this preprocessing tactic, we are enabling automated reasoning across a significant portion of goals typical of machine code analysis, thus reducing the time, effort, and expertise required to do significant verification.**

***Index Terms*—Binary Analysis, SMT Solvers, Interactive Theorem Provers**

I. Introduction

Maintainers of mission-critical systems often seek out formal methods for their rigorous techniques of ensuring correct behavior. However, the rigor of these techniques leads them to be expensive, requiring hours of expert work to produce useful results. This motivates automating formal methods, allowing such techniques to be more accessible and quicker to implement.

A significant amount of formal analysis is dedicated to machine code, as it represents exactly what runs on real-world systems. Despite this, existing analysis tools either lack the robustness and dependability of ISA semantics formalized in an interactive theorem prover (ITP) or lack significant automation. The latter applies to *Picinæ*, leading to significant time, effort, and expertise dedicated to proving theorems that SMT solvers quickly solve.

This motivates automating the goals that typically show up in Picinæ analysis with *SMTCoq*. However, this comes with two challenges. First, Picinæ contains abstractions concerning memory and bit-properties that don't have clear direct equivalents *SMT-LIB*. Second, SMTCoq uses a bounded, dependently typed bit-vector, which we call *BV*, while Picinæ uses Rocq's N type, which is non-dependently typed and unbounded. To solve this, we are working on a tactic that lowers Picinæ's abstractions into basic bit-arithmetic and converts Rocq's N type to SMTCoq's BV.

II. Background

A. Picinæ

Picinæ is a framework within the Rocq interactive theorem prover for representing and reasoning over arbitrary machine code [1]. Picinæ is able to prove any properties that are parameterized by a history of CPU states, such as correctness, timing [2], and fault tolerance [3].

It primarily uses Rocq's N type, an unbounded, unsigned bit-vector, to reduce redundancy by allowing the user to specify and prove properties that apply across differently sized constructs. To represent bounded arithmetic over `N` of bit-width sz, Picinæ encodes overflow via $mod\ 2^{sz}$ and underflow via $-_{\mathrm{sz}}$. Picinæ provides abstractions that make reasoning about machine code more intuitive for the user, though with the drawback that SMT solvers don't understand them. These abstractions typically concern memory, strings, and bit properties, and a subset are shown below:

1) $mem\big[pos\,..\,pos+len\big] \leftarrow input$ — write len bytes from $input$ into mem at byte address pos of size $addrsize$.
2) $mem\big[pos\,..\,pos+readlen\big]$ — read $readlen$ bytes from mem at byte address pos, returning a natural number.
3) $\mathsf{LastEq}_w(len, i)$ — the last index $j < len$ with $j \equiv i \pmod{2^w}$.

B. SMTCoq

SMTCoq allows for communication between Rocq and SMT/SAT solvers [4]: VeriT [5], CVC4 [6], and ZChaff [7]. It supports propositional logic, functional arrays with extensionality, linear integer arithmetic, and uninterpreted function symbols, though only the theory of fixed size bit-vectors is relevant to this paper. SMTCoq's main tactic `smt` consists of two primary components that enable this: the reifier and the proof checker.

The reifier converts a subset of Rocq's goals into equivalent SMT-LIB, defaulting to uninterpreted function symbols if it does not recognize a expression. In the case of fixed size bit-vectors, SMTCoq supports reifying all expressions within a locally defined bit-vector module that contains equivalents to a subset of expressions in SMT-LIB.

After reification, the generated SMT-LIB is processed by a proof witness producing SMT/SAT solver that supports the relevant theories; CVC4 and VeriT in the case of bit-vectors. When the SMT solver produces an unsat result, it is also paired with a proof witness in a proof-format; LFSC [8] for CVC4

https://doi.org/

and Alethe [9] for VeriT. SMTCoq then verifies the soundness of the proof witness using its formally verified proof checker; if no errors in logic are found, then the original Rocq goal is automatically solved.

III. Technical Approach

The preprocessing tactic, named `preprocess`, will contain 3 tactics sequentially: `bit_blast`, `canonicalize`, and `N_to_BV`. The `bit_blast` tactic enumerates all in-bound bits, also known as *bit blasting*, while `canonicalize` and `N_to_BV` sequentially converts N to BV; `canonicalize` rewrites Picinæ's abstractions to basic bit-arithmetic and `N_to_BV` turns the resulting expressions into equivalent operations within SMTCoq's BV module.

This combination of tactics converts a typical Picinæ goal into a format that can be discharged to SMTCoq's `smt` tactic, thus automating many goals common in machine code analysis.

A. `bit_blast`

`bit_blast` expands a goal into separate goals about each bit that lies within bounds. It is similar to the typical bit-blasting found in SMT solvers in that it reduces goals to bit-level reasoning, though different in that it does not produce a pure boolean circuit. The N type has no explicit bounds, so the `bit_blast` tactic leverages implicit bounds introduced by operations such as $-_{\text{sz}}$ and $\bmod 2^{sz}$; x_i represents the bit of x at index i.

$$x \bmod 2^w = y \bmod 2^{w'} \;\leftrightarrow\; (i < \max(w, w') \;\rightarrow\; x_i = y_i)$$

After enumeration, it calls the `canonicalize` tactic on each generated goal, concluding the `bit_blast` tactic.

While this tactic can't reason about truly unbounded N values, this doesn't apply to most goals typical of Picinæ analysis. However, Picinæ goals may use other methods of denoting bounds rather than $\bmod\ 2^i$, so future work concerning this tactic involves allowing it to infer the bounds based on hypotheses and other operations.

B. `canonicalize`

The `canonicalize` tactic simplifies complex, abstract functions into bit-arithmetic/logic expressions using *spec* theorems. Below are theorems required to canonicalize $mem\big[pos\,..\,pos+len\big] \leftarrow input$ and $mem\big[pos\,..\,pos+readlen\big]$.

$$(mem\big[pos\,..\,pos+len\big] \leftarrow input)_i = \begin{cases} v_{\text{final\_eqm}(w+3,\, len\cdot 8,\, i-a\cdot 8)} \\ \quad \text{if } (i - a\cdot 8 < len\cdot 8)\ \&\&\ (i < 2^{w+3}) \\ m_i \quad \text{otherwise} \end{cases}$$

$$mem\big[pos..pos+len\big]_i = (i < 8len)\ \&\&\ m_{(8a+i \bmod\ 2^{w+3})}$$

$$\mathsf{LastEq}_w(len, i) = len - 2^w + (i - \max(len,\, 2^w)) \bmod 2^w$$

$$(x\ -\ y) \bmod 2^{sz} \quad = \quad \begin{cases} x -_{\text{sz}} y & \text{if } x > y \\ 0 & \text{otherwise} \end{cases}$$

$$x_n = \big(x >> n\big)\ \&\ 1$$

The resulting goal is typically large and unreadable, but primitive enough for `N_to_BV` to process.

While `canonicalize` handles most abstractions concerning memory and bit-properties, there are more advanced abstractions concerning strings it does not take into account as of yet. Future work involves supporting these abstractions.

C. `N_to_BV`

Once the goal has been normalized, `N_to_BV` exhaustively converts N to BV, where $\lceil x \rceil_{sz}$ constructs a $BV\ sz$ from $x \in \mathbb{N}$, recovering the size that N does not track. The first rewrite is:

$$\varphi(x \bmod 2^{sz},\ y \bmod 2^{sz}) \iff \varphi(\lceil x \rceil_{sz},\ \lceil y \rceil_{sz})$$

where φ ranges over the supported comparison operators ($=$, $\leq$, $<$, $\geq$, $>$). These rewrite rules cover virtually all relevant bit-arithmetic, and a subset of them are shown below:

$$\lceil x + y \rceil_{sz} = \lceil x \rceil_{sz} +_{\text{BV}} \lceil y \rceil_{sz}$$

$$\lceil x\ \&\ y \rceil_{sz} = \lceil x \rceil_{sz}\ \&_{\text{BV}} \lceil y \rceil_{sz}$$

$$\lceil x \bmod 2^{sz'} \rceil_{sz} = \begin{cases} \lceil x \rceil_{sz} & \text{if } sz' \geq sz \\ \lceil x\ \&\ \mathbf{1}_{sz'} \rceil_{sz} & \text{otherwise} \end{cases}$$

$$sz' \geq sz \;\rightarrow\; \lceil x -_{sz'} y \rceil_{sz} = \lceil x \rceil_{sz} -_{\text{BV}} \lceil y \rceil_{sz}$$

Currently, `N_to_BV` supports virtually all bit-arithmetic operations, so all bit-arithmetic goals can be rewritten to a form that can almost be processed by SMTCoq. Fully processing these goals requires modifying SMTCoq's reifier so that it can infer the bounds of a quantified N variable and produce equivalent SMT-LIB.

IV. Conclusion and Future Work

This paper presents in-progress work aimed at integrating SMTCoq with Picinæ by converting Picinæ's abstractions into SMTCoq's primitive bit-vectors using 3 tactics: `bit_blast`, `canonicalize`, and `N_to_BV`. By creating this tactics, we aim to automate tedious analysis only available to domain experts, thus increasing the availability of formal methods.

Current work is aimed towards the creation of the `bit_blast` tactic. Meanwhile, `canonicalize` can handle most abstractions concerned with memory so future work is focused on covering all of Picinæ's abstractions, especially concerning strings. The most complete of the 3 tactics is `N_to_BV`, as it support virtually all bit-arithmetic, though the resulting goal can only be processed by a modified version of SMTCoq.

After the creation of these tactics is complete, subsequent work aims to benchmark SMTCoq to evaluate whether it correctly assesses goals produced by `preprocess`. This benchmarking can help inform the design of these tactics, since SMTCoq may handle certain goals more effectively than others.

REFERENCES

[1] K. W. Hamlen, D. Fisher, and G. R. Lundquist, "Source-free Machine-checked Validation of Native Code in Coq," in *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, (London United Kingdom), pp. 25–30, ACM, Nov. 2019.

[2] C. Averill, "Formally-Verified, Tight Timing Constraints for Machine Code," *PLDI SRC*, 2025.

[3] C. Averill, I. Buzzetti, A. Bellon, and K. Hamlen, "LAPSE: Automatic, formal fault-tolerant correctness proofs for native code," *NDSS BAR*, Feb 2026.

[4] B. Ekici, A. Mebsout, C. Tinelli, C. Keller, G. Katz, A. Reynolds, and C. Barrett, "Smtcoq: A plug-in for integrating smt solvers into coq," in *Computer Aided Verification* (R. Majumdar and V. Kunčak, eds.), (Cham), pp. 126–133, Springer International Publishing, 2017.

[5] T. Bouton, D. Caminha B. de Oliveira, D. Déharbe, and P. Fontaine, "verit: an open, trustable and efficient smt-solver," in *International Conference on Automated Deduction*, pp. 151–156, Springer, 2009.

[6] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "cvc4," in *International Conference on Computer Aided Verification*, pp. 171–177, Springer, 2011.

[7] Y. S. Mahajan, Z. Fu, and S. Malik, "Zchaff2004: An efficient sat solver," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 360–375, Springer, 2004.

[8] A. Stump, D. Oe, A. Reynolds, L. Hadarean, and C. Tinelli, "Smt proof checking using a logical framework," *Formal Methods in System Design*, vol. 42, no. 1, pp. 91–118, 2013.

[9] H.-J. Schurr, M. Fleury, H. Barbosa, and P. Fontaine, "Alethe: Towards a generic smt proof format," *arXiv preprint arXiv:2107.02354*, 2021.