ECE653 Final Course Project

Zhiyuan Ning, 21050449, Vladyslav Yusiuk, 21050623, and Aiswarya Tom, 21051073

University of Waterloo - Electrical and Computer Engineering, Ontario, Canada

Abstract. Final Project of ECE653 course - choice 0

1 Implementation of Bit-Vector in Symbolic Execution Engine

1.1 Fundamentals of Bit-Vector Implementation in Symbolic Execution

The integration of bit-vector theory into our symbolic execution engine is rooted in the fundamental need to accurately model and analyze software at the bit level, particularly pertinent in low-level programming and hardware design. This section of the report delves into the underlying principles and the significance of bit-vectors in the realm of symbolic execution.

Bit-vector theory operates on the premise that data is represented as vectors of bits, rather than traditional high-level data types. This representation is akin to how data is managed within computer hardware, allowing for a more precise emulation of bitwise operations. Each bit-vector carries a specified bit-width, corresponding to the data sizes typically encountered in machine-level operations (e.g., 8-bit, 16-bit, 32-bit, 64-bit). This approach facilitates the accurate modeling of overflows, underflows, and bitwise manipulations that are inherent in low-level programming languages and hardware instruction sets.

In symbolic execution, rather than executing program code with concrete inputs, the engine explores program behavior across a range of possible inputs. When bit-vectors are employed, the engine symbolically represents variables as bit-vectors, and their interactions are modeled as operations on these bit-vectors. This method is particularly effective in uncovering edge cases and subtle bugs that are dependent on specific bit-level operations, which might be overlooked when using higher-level abstractions.

Bit-vectors enable the symbolic execution engine to model and reason about the program's behavior with a high degree of accuracy, particularly in scenarios involving bitwise operations and low-level data manipulation. By closely mirroring the operations of low-level programming, bit-vectors make it possible to thoroughly analyze systems programming code and firmware, where traditional abstract data types fall short. Bit-vector support empowers the engine to handle complex operations like bit shifts, rotations, and bitwise arithmetic, which are crucial in embedded systems, cryptographic algorithms, and system utilities.

In summary, the integration of bit-vector theory into our symbolic execution engine enhances its ability to perform in-depth and precise analysis of software, especially in areas where low-level data manipulation and hardware interactions are prevalent. This advancement not only improves the engine's capabilities in current scenarios but also opens avenues for future enhancements and applications in more complex software analysis tasks.

1.2 Implementation of Bit-Vectors in the Symbolic Execution Engine

Integrating bit-vector support into our existing symbolic execution engine, particularly within the 'sym.py' module, required a series of meticulous codebase enhancements to accommodate the complexities and nuances of bit-level operations. This critical development phase focused on embedding the bit-vector theory into the core functionalities of our engine, ensuring that it could accurately handle and analyze low-level program behaviors.

The foundational step was the redefinition of the state environment within the engine. Variables, which were previously represented using generic data types like integers or booleans, were now modeled as bit-vectors. This shift necessitated the creation of bit-vector variables with specific bit-widths to accurately represent the data types encountered in low-level programming.

The engine's capability to process symbolic operations was extended to include bit-vector specific operations. Functions and methods within 'sym.py' were adapted to handle a range of bitwise operations such as AND, OR and NOT. This change was important for the engine to simulate the exact behavior of low-level code.

A critical aspect of the implementation was the integration and configuration of the Z3 solver to handle bit-vector formulas and values. This involved ensuring that the solver could efficiently interpret and process the constraints and operations defined by the bit-vectors.

The following visitor methods were modified to incorporate bit-vectors: visit_BoolConst(), visit_IntConst(), visit_RelExp(), visit_BExp(), visit_AsgnStmt(), and visit_HavocStmt(). For visit_BoolConst(), visit_IntConst(), we used z3.BitVecVal instead of z3.Int to represent the value. The int constant values are 32 bits long while the boolean values are 1 bit long since it only needs to represent true or false and true will be converted to 1 and false will be converted to 0. For variables (x, y, etc) in visit_AsgnStmt() and visit_HavocStmt(), we used BitVec since they were not constants. After we changed the interpretation of the variables and values, we started modifying expressions in our symbolic execution engine. For visit_RelExp(), we used ULE, ULT, UGE and UGT for "¡=", "¡", "¿=" and "¿", respectively, but we could not find a function for equality so "=" remained the same. In our visit_BExp() implementation, "NOT" were changed to "$\tilde{}$", "AND" were interpreted by "&" and "OR" were represented by "—". They are all bit-wise operators. These summarizes all of our changes made to incorporate bit-vectors in our symbolic execution engine.

To test our implemenatation, we added extra test cases to reach both statement and branch coverages. We were more focused on expressions, which included boolean, arithmetic and real expressions since they were where most of our modifications took place. We also included more if and while statements since the aforementioned expressions were inside these two statements. Finally, we added more assert and assume statements since the expressions were also used in these places and they were easy to debug and test.

The successful integration of bit-vectors into our symbolic execution engine significantly elevates its analytical precision, particularly in scrutinizing low-level code and hardware-related operations and this is a crucial part in many real life scenarios and applications including cryptos. The codebase enhancements have enabled the engine to handle low-level code and representation of the symbolic engine.

Looking forward, we anticipate further refining this implementation. Potential enhancements include advanced bit-vector manipulation techniques, improved performance tuning for large-scale applications, and the exploration of hybrid symbolic execution strategies that synergize bit-vector analysis with other symbolic execution paradigms. These ongoing developments aim to keep our engine at the forefront of software analysis technology, equipped to tackle the evolving challenges of software verification and analysis.

1.3 Challenges Encountered During Bit-Vector Implementation in Symbolic Execution

The implementation of bit-vector theory into our symbolic execution engine was a complex and intricate process, marked by a range of challenges that tested the limits of our technical expertise and problem-solving abilities. One of the primary challenges was the integration of bit-vectors into the existing framework, which necessitated significant modifications to our state environment. Adapting the engine to accurately represent and manipulate variables as bit-vectors, instead of traditional data types, involved deep changes at the core level of our codebase. This required not only a theoretical understanding of bit-vectors but also practical skills in effectively modifying the engine's architecture.

Another major challenge was the extension of the engine's capabilities to handle a variety of bitwise operations. Ensuring that these operations were accurately simulated in a symbolic context demanded a meticulous approach to coding and testing. Bitwise operations, such as AND, OR, XOR, and bit shifts, introduced a level of complexity that was previously unaddressed in our engine. Each of these operations had to be implemented in a way that mirrored their behavior in low-level programming, adding layers of complexity to the engine's functionality.

The integration with the Z3 solver posed its own set of challenges. Configuring the solver to efficiently process and interpret bit-vector formulas was critical to the success of the implementation. This required a nuanced understanding of how the solver processes symbolic constraints and an ability to optimize its

performance for bit-vector operations, especially considering the potential for increased computational demands.

The integration of bit-vector theory into our symbolic execution engine presented a complex range of issues that went beyond programming and theoretical knowledge. Teaching and training our development team on the intricacies of bit-vector theory and its applications in symbolic execution was a major challenge. To provide team members with the necessary skills, this required planning specific training sessions, workshops, and knowledge-sharing efforts. Closing the knowledge gap was essential to guaranteeing that each team member could properly contribute to the implementation process.In order to handle the complexities of bit-vector integration, team members with varying backgrounds and levels of experience needed to work together in an environment that promoted collaborative problem-solving. Developing a culture of continuous learning and overcoming the difficulty of knowledge distribution were crucial in creating a cohesive and competent team that could successfully navigate the difficulties of bit-vector implementation.

Performance optimization was a significant hurdle throughout the implementation process. The introduction of bit-vectors naturally led to an increase in computational complexity, particularly in terms of memory usage and processing power. Addressing this involved not only optimizing the representation of bit-vectors within the engine and solver but also implementing caching mechanisms for frequently executed operations to minimize overhead.

Finally, the testing and validation phase presented its own set of challenges. Developing robust test cases that effectively captured the nuances of bit-vector operations was crucial. These tests needed to be comprehensive enough to cover a wide range of scenarios, ensuring that the engine's updated capabilities were rigorously evaluated and validated. Ensuring that these tests accurately reflected the complexities of real-world bit-level manipulations was paramount to confirming the reliability and accuracy of the bit-vector implementation.

Moreover, the documentation difficulty increased the project's complexity even further. It was essential to keep thorough and current documentation reflecting the modifications to the engine's architecture, APIs, and features as bit-vector implementation progressed. The provision of unambiguous and easily readable documentation was crucial in promoting teamwork, supporting troubleshooting, and serving as a valuable asset for upcoming maintenance and development initiatives. It took constant work to strike the right balance between technical detail and readability in the documentation so that developers, both inside and outside the team, could use it to their advantage to comprehend, put into practice, and expand upon the capabilities of the symbolic execution engine with bit-vector support.

In summary, the journey to integrate bit-vector theory into our symbolic execution engine was fraught with challenges that spanned from deep architectural changes to performance optimization and complex problem-solving in coding and testing. Overcoming these challenges was a testament to our team's tech-

nical acumen and dedication, ultimately leading to a more sophisticated and capable symbolic execution engine.

1.4 Conclusion: Elevating Symbolic Execution with Bit-Vector Theory

The integration of bit-vector theory into our symbolic execution engine marks a pivotal advancement in our capabilities for software analysis and verification. This enhancement has not only solidified our engine's proficiency in handling and analyzing low-level code, but it has also established a new benchmark for precision in the realm of symbolic execution.With this significant improvement in accuracy and adaptability, our symbolic execution engine is now at the forefront of the field's most advanced software analytical instruments. Bit-vector theory integration enables our engine to decipher complex and hitherto unsolvable parts of low-level code, providing new opportunities for more in-depth understanding of program behavior. Our engine has successfully overcome the challenges associated with bit-vector manipulation, providing a more sophisticated comprehension of binary-level operations. This allows it to identify minute interactions that could have been missed in conventional symbolic execution methods.

Adding bit-vector theory to our symbolic execution engine makes it more accurate and establishes it as a valuable resource for developers and security experts. This development reinforces our dedication to offering cutting-edge software validation and verification solutions, boosting trust in the dependability and security of software systems.Moving forward, bit-vector integration creates the foundation for further developments in symbolic execution techniques, indicating ongoing progress toward our goal of deciphering the complexities of software behavior and improving the reliability of digital systems. Essentially, this integration is a big step toward a more advanced and reliable software analysis era, and our symbolic execution engine is leading the way in this revolutionary process.

The adoption of bit-vectors significantly enhances the precision and reliability of our engine. It enables us to accurately model and analyze software at the bit level, which is particularly vital in systems programming and hardware design where bit-level operations are crucial. The broadened scope of our engine, with its newfound ability to handle complex bitwise manipulations, allows us to delve deeper into the intricacies of low-level software behavior. This advancement is instrumental in improving the safety and reliability of software, especially in critical systems where subtle bit-level issues can have far-reaching consequences.

Reflecting on the implementation process, the journey to integrate bit-vectors was both challenging and enlightening. It demanded not only a deep understanding of the theoretical aspects of bit-vectors but also a keen insight into practical implementation strategies. This process highlighted the critical importance of comprehensive testing and the need for performance optimization, particularly when addressing the complexities associated with bit-vector operations.

The lesson learned from the experience was the importance of flexibility and iterative development while dealing with changing obstacles. The robustness of

our bit-vector implementation was refined through iterative testing and improvement cycles, which also yielded important insights into potential weaknesses and corner cases. Additionally, the integration process made clear how crucial it is for team members to communicate and work together, as these skills were critical to overcome complex obstacles.The lessons we've gained from integrating bit-vectors will surely impact how we tackle projects in the future. Our team now feels more resilient and determined than ever thanks to the experience, which has also established a standard for approaching difficult technical problems strategically and cooperatively. To put it simply, bit-vector integration was a major step forward for our symbolic execution engine and a foundation for ongoing innovation and evolution inside our development processes.

Looking to the future, there are several promising directions for further enhancing our engine. Exploring more sophisticated bit-vector manipulation techniques could open new avenues in areas such as cryptographic analysis and embedded systems. The potential for hybrid execution strategies, which combine bit-vector analysis with other symbolic execution paradigms, could lead to a more versatile and powerful tool, adaptable to various software types. Moreover, as the capabilities of our engine expand, continuous efforts in performance optimization will be crucial, ensuring efficiency, especially when analyzing large-scale software systems.

In conclusion, the integration of bit-vector theory into our symbolic execution engine is more than a mere technical upgrade; it represents a stride towards more advanced, reliable, and thorough software analysis. As we continue to refine our engine and expand its capabilities, our commitment remains steadfast in keeping pace with the evolving landscape of software development and analysis, ensuring that our tools remain relevant and effective in the face of changing technological challenges.Furthermore, our symbolic execution engine's incorporation of bit-vector theory is evidence of our steadfast dedication to accuracy and dependability in software analysis. This improvement is a dynamic statement of our continued quest for greatness rather than a static accomplishment. With bit-vector support added, our symbolic execution engine is now more ready to adapt to and thrive in the face of new obstacles as we negotiate the always shifting world of software development.

This integration is important because it shows our dedication to serving the larger research and practitioner communities, which goes beyond our short-term growth objectives. Through the adoption and incorporation of novel theories such as bit-vectors, we contribute to the development of symbolic execution techniques. This cooperative effort promotes a community-driven quest of software analysis excellence by extending an invitation for discussion and knowledge sharing.To put it simply, bit-vector theory's incorporation is evidence of our commitment to remain at the forefront of technical development. It highlights our proactive effort to not only adhering to industry standards but also instigating ground-breaking ideas that advance the software analysis profession. By means of this integration, we strengthen our symbolic execution engine and further advance the joint endeavor of software engineering and analysis excellence.

2 Use incremental solving mode of Z3-Introduction

A key component of software analysis is symbolic execution, which uses symbolic expressions to represent variables and allows for a methodical investigation of program pathways. Even though symbolic execution works well, it has issues caused by re-evaluating the same constrains over and over again for different paths. The computing inefficiencies caused by these redundancies reduce the overall effectiveness and efficiency of symbolic execution.

Z3, a cutting-edge solver of Satisfiability Modulo Theories (SMT), offers incremental solving capabilities in answer to this difficulty. With the help of the incremental solving function, restrictions can be added and removed dynamically while the problem is being solved without having to start from scratch. In Z3, the main goal of incremental solution is to maximize symbolic execution performance by strategically controlling program path exploration and reducing unnecessary computations. Z3 provides at least two independent incremental solution interfaces: Scopes and Assumptions. And in our implementation, we chose the Scopes approach.

2.1 Incremental Interfaces in Z3

Scopes Interface Z3's Scopes interface allows you to organize sets of assertions within a solution. Each scope denotes a point in the symbolic execution at which a set of restrictions is added or removed. This acts like a stack which has a pop() and push() method that dynamically add or remove an item (in this case is the constraints or formulas) and enables more fine-grained control over the solution process. Symbolic execution tools that use the Scopes interface can define scopes at any moment during program execution. Constraints added within a scope are only taken into account within that scope, providing a confined context for symbolic execution. Scopes allow the symbolic execution engine to selectively add and remove constraints, giving it more control over the solution process. The Scopes interface simplifies the handling of context-specific constraints, potentially boosting overall symbolic execution efficiency.

Assumptions Interface Another technique Z3 provides is the Assumptions interface. It allows the symbolic execution engine to check satisfiability under the assumption of a set of literals. These assumptions can later be asserted or withdrawn, giving flexibility in regulating the set of restrictions during symbolic execution. During symbolic execution, assumptions can be utilized to assist the solver without committing to explicit restrictions. This enables the symbolic execution engine to investigate multiple paths without having to explain limitations again.

Assumptions allow the symbolic execution engine to adapt to multiple execution pathways by providing a dynamic approach to steer the solver's investigation. The Assumptions interface reduces the need to re-specify restrictions, which contributes to increased symbolic execution efficiency.

2.2 Interface Scopes: A Methodical Approach to Constraint Handling

Sets of assertions can be arranged systematically within a solution using the Scopes interface in Z3. The symbolic execution engine can establish unique scopes at various stages of the execution path thanks to the Scopes interface, in contrast to a conventional solving method. During the study of a program, symbolic execution tools can be used to intentionally establish scopes that encapsulate sets of restrictions at different points. Constraints are organized and managed effectively within each scope since they are only taken into account within that particular context.

With the use of scopes, symbolic execution engines can add and remove constraints at predetermined program execution locations. Every scope provides a framework within which constraints can be managed, providing an organized and specific method for handling the changing collection of constraints. During symbolic execution, the Scopes interface helps to minimize resource use by restricting the evaluation of constraints to particular scopes.

2.3 Implementation

We chose Scopes interface to implement solver incrementality. What we have changed are totally different than our bit-wise implementation. To incorporate Scopes, we utilized the solver's push() and pop() methods whenever the program diverts. For example, in the if statement, we did push when we wanted to explore the else branch and in the while statement (no invariants), we also did it when the condition evaluated to false. The while loop with invariants are similar to the if statement although it had an assert statement when converted to the non-loop representation. The last visitor method we changed was for the assert statement and it was done using the similar fashion.

There was one thing that we took extra care of and it was all the variables the Symbolic State kept track of. For example, we first needed to remember all the current values before we encounter the if, while or assert statement. And then we invoked push() method and after we finished exploring the other path, we had to restore all the variables, which means we need to remove the variables saved in the other path. Doing this ensures that we do not save unavailable variables for the original path.

To test our implementation for the Scopes interface, we added more tests that used if, while and assert statements since there will be multiple paths available to explore in these statements. We looked into the running program step by step to check if the variables were correctly removed after the pop() method call and if the new state's constraints were properly discarded after returning to the original state. Finally, we achieved statement and branch coverage.

Symbolic execution engines can d incremental solving to maximize the solving process by using the Scopes interface. Performance can be enhanced by the symbolic execution engine by avoiding duplicate solving by utilizing previously solved constraints and gradually building upon them.The engine's capacity to

dynamically modify constraints while it is running facilitates effective path exploration.The Scopes interface is a well-organized and potential tool that provides a methodical approach to managing and organizing restrictions for symbolic engine. It also contributes to increased performance and efficiency by enabling a more contextualized and regulated exploration of program paths.

2.4 Conclusion

The incremental solution capabilities of Z3,the Assumptions and scope have an important and revolutionary aspect to the performance of symbolic execution.These interfaces provide distinct benefits for improving the efficacy and efficiency of symbolic analysis, including fine-grained control and adaptive constraint handling.By the flexibility in selecting between Assumptions and Scopes it has the ability to customize to fit the unique requirements of the symbolic execution process.This represents an important leap in tackling performance concerns and opens up new paths for more and more complex analysis.

The exact control and contextual management feature , the Scopes interface has the ability to transforms the way constraints are handled in symbolic execution engines, resulting in significant improvements in reliability and efficiency. This aspect gives engines the ability to customize constraint-handling algorithms, allowing for targeted program path exploration by generating distinct scopes at various phases of execution. Using the Scopes interface in particular to solve problems incrementally has numerous important benefits.This method minimizes duplication and improves resource efficiency by enabling the selective insertion and removal of limitations. Contextualizing restrictions inside scopes makes it easier for symbolic execution to be more ordered and organized. Because of its flexibility, the Scopes interface can be used to investigate different approaches and dynamically adjust to changing program needs.

Present issues are resolved and future innovations are made possible by integrating the Scopes interface into symbolic execution engines, . The use of incremental solving with scopes lays the base for future developments, such hybrid strategies that integrate several approaches to solving problems or heuristic-driven strategies like A* search. Use of advanced state management and exploration techniques due to the dynamic nature of the Scopes interface is required for future advancement and development.

An important change from the previous bit-wise approach was the introduction of the Scopes interface for solver incrementality. Among the changes were the deliberate use of the solver's push() and pop() functions inside of particular program structures, such as while loops and if statements. The exact information was preserved during program diversions and the subsequent restoration to the original state due to management of variables within the Symbolic State. Thorough testing, including more scenarios including if, while, and assert statements, verified the implementation of the Scopes interface and confirmed its dependability and effectiveness in examining various program pathways.

In conclusion we can say.the addition of incremental solution via the Scopes interface is a major advancement in symbolic execution. It effectively addresses

problems and issues associated with repetitive searches and computing inefficiencies by utilizing more versatile and evolving methods for symbolic program path exploration. This tactical use not only enhances symbolic execution engines' capabilities but also provide significant developments in the rapidly growing field of software analysis.

References

1. Gurfinkel, A. (2022). Symbolic Execution Semantics for WHILE Language.
2. Prof. Arie Gurfinkel, F.: Verification Condition Generation. Lecture Slides, 1–30 (2019)
3. Nielson, H. R., amp; Nielson, F. (1999). Semantics with applications: A formal introduction. J. Wiley.
4. Bradley, A. R., amp; Manna, Z. (2010). The calculus of computation: Decision procedures with applications to verification. Springer.
5. Schoning, U. (1989). Logic for computer scientists. Birkhauser.
6. Kroening, D., amp; Strichman, O. (2018). Decision procedures an algorithmic point of view. Springer Berlin.