





# Poster: A Unit Proofing Framework for Code-level Verification

Paschal C. Amusuo   
Purdue University  
pamusuo@purdue.edu

Parth V. Patil   
Purdue University  
patil185@purdue.edu

Owen Cochell   
Michigan State University  
cochello@msu.edu

Taylor Le Lievre   
Purdue University  
tlelievr@purdue.edu

James C. Davis   
Purdue University  
davisjam@purdue.edu

**Abstract**—Formal verification provides mathematical guarantees that a software is correct. Design-level verification tools ensure software specifications are correct, but they do not expose defects in actual implementations. For this purpose, engineers use code-level tools. However, such tools struggle to scale to large software. The process of “Unit Proofing” mitigates this by decomposing the software and verifying each unit independently. We examined AWS’s use of unit proofing and observed that current approaches are manual and prone to faults that mask severe defects. We propose a research agenda for a unit proofing framework, both methods and tools, to support software engineers in applying unit proofing effectively and efficiently. This will enable engineers to discover code-level defects early.

**Index Terms**—Vision, Formal methods, Empirical software eng.

## I. INTRODUCTION

Software underlies many critical digital and cyber-physical systems. Code-level defects, such as arithmetic errors and memory corruption vulnerabilities, compromise system safety and security and cause significant losses [1], [2]. Traditional engineering practices, such as program analysis [3] and runtime sandboxing [4], can detect or mitigate such defects. However, these methods validate software only approximately and cannot guarantee correct behavior on all inputs. In contrast, formal verification [5] proves software correctness, guaranteeing no code-level defects. This raises an important (and perennial) question [6]: how to make formal verification more cost-effective for software engineers?

Engineering tools and processes help reduce the cost of applying a technology. For formal methods, there are code-level verification tools, *e.g.*, the C and Rust bounded model checkers (CBMC [7], Kani [8]). To improve both engineering and tool scalability, engineers decompose the software and verify individual functions within it via *unit proofs* [9]. These unit proofs model the function’s interaction with other software components and verify the function using this model. We refer to the process of developing, using, and maintaining unit proofs as *unit proofing*. Organizations such as AWS [10] and ARM [11] have adopted unit proofing to ensure software correctness and memory safety.

Since code-level verification depends on the unit proofs, they must be correct. However, there is limited guidance and tool support for constructing unit proofs. Existing guidelines are *ad hoc* and require expertise [12]. There is little publicly-available tool support, leading to costly manual development of unit proofs. To assess the quality of current practices, we

evaluated 11 unit proofs used to verify vulnerable functions in a widely-used operating system. While 5 successfully identified the corresponding vulnerabilities, the remaining 6 contained faults that obscured others.

We believe that empirically-based guidelines and enhanced tool support will make unit proofing easier and enable early discovery of code-level defects. We propose a research agenda for a principled end-to-end unit proofing framework that can support software engineers in each stage of the unit proofing process. The proposed framework comprises of methods and tools and will help software engineers decompose a software into optimal verifiable units, develop correct unit proofs for each unit, validate and repair faulty unit proofs, and ensure unit proofs are updated as the software evolves. We identify eight open questions that will need to be addressed, and share our future plans to this end. If the proposed unit proofing framework helps software engineers verify their software code during development, then it will disrupt existing defect-finding practices such as static analysis, unit testing, and fuzzing.

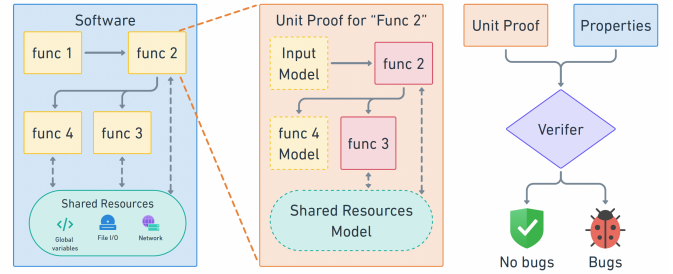


Fig. 1: Overview of Unit Proofing. To verify function 2, an engineer models its input, shared resources, and other functions. A verifier can then check it in isolation.

## II. EXPLORATORY STUDY OF UNIT PROOFING

Many questions arise when considering any engineering process, such as “Does it work?” and “What does it cost?” [13]. **While there are empirical evaluations of code-level verification tools on software benchmarks [14], no studies address their effectiveness on real software or within the unit proofing process.** As a first step, we ask:

*RQ: Do existing unit proofs expose security vulnerabilities?*

We evaluated this question using 11 CVEs in FreeRTOS and the unit proofs that target the vulnerable functions [15].

Table I: Assessing unit proofs on known vulnerabilities. Only 5 of the 11 vulnerabilities were exposed by the existing FreeRTOS unit proofs. *UP*: Unit Proofing.

Recreated Vuln	Exposed	Reason	UP Stage
CVE-2018-16523	No	Insuff. properties	Design
CVE-2018-16524	No	Function Modeling	Development
CVE-2018-16525	No	Out-of-date Proofs	Maintenance
CVE-2018-16527	No	Insuff. Input Model	Development
CVE-2018-16600	No	Insuff. Input Model	Development
CVE-2018-16603	No	Insuff. Input Model	Development
CVE-2021-31571	Yes	—	—
CVE-2018-16526	Yes	—	—
CVE-2018-16599	Yes	—	—
CVE-2018-16601	Yes	—	—
CVE-2018-16602	Yes	—	—

### A. Methodology

For each vulnerability, we identified and removed any validations introduced in the function to fix the vulnerability. We then executed the unit proofs using CBMC v5.95.1 and assessed whether the vulnerability was reported. If not, we identified the reason and repaired the unit proofs as needed.

### B. Results

Only 5/11 recreated vulnerabilities were exposed by FreeRTOS unit proofs (Table I). With changes to the unit proofs, all vulnerabilities were exposed and one new high-severity vulnerability found (CVE-2024-38373). This result shows that unit proofs are effective, but unit proof engineers can err. Further details can be found in our technical report [16].

## III. RESEARCH AGENDA: A UNIT PROOFING FRAMEWORK

This section presents a unit proofing framework, comprising novel tools, to automate unit proof development, repair and maintainance. We discuss the unit proofing stages and present a research plan to automate them (Figure 2).

1) *Software Decomposition*: Our goal is to decompose a software into the minimum number of solvable units.

*Software Decomposer*: Optimal software decomposition will require new methods to estimate model complexity and verification time. Building on research linking program features to specification size and effort [17], [18], we will analyze unit proofs to understand how program features influence complexity and duration. These insights will guide a cost estimator design that evaluates function features to estimate complexity and verification time and a decomposer algorithm to iteratively replace the most expensive functions with models.

2) *Unit Proof Design and Development*: Our goal is to develop unit proof with models that precisely represent the unit’s interactions with other software components.

*Proof Builder*: Deriving software models is a known challenge. Traditional learning-based assume-guarantee reasoning [19], [20] refines assumptions iteratively using counterexamples, but applying this to functions with multiple interactions is costly due to the need for repeated model

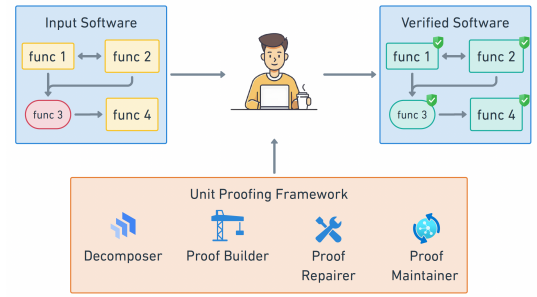


Fig. 2: An end-to-end agenda for unit proof engineering. Software engineers use a set of tools and intelligent agents to verify the memory safety of applications.

checking [21]. The proposed proof builder will use a property-violation guided learning approach to generate the *weakest precondition models* for a target function. It will start with generic proof models and iteratively refine the models until all reachable code is covered and reported violations are resolved. The learned models will then be validated for correctness.

3) *Unit Proof Repair*: Our goal is to detect and repair unit proof faults, such as those that mask vulnerabilities (§II).

*Proof Repairer*: First, we will develop a taxonomy of unit proof faults and corresponding resolutions by empirically studying existing and new unit proofs, assessing their coverage, violation reports, and bug detection abilities, and identifying the different faults that occur and their resolutions. Using the developed taxonomy, we will design the proof repairer by combining the fast but imprecise program analysis techniques with the code reasoning abilities of large language models (LLMs) [22], [23].

4) *Unit Proof Evolution and Maintenance*: The goal here is to automatically update unit proofs as the software changes.

*Proof Updater*: When the software changes, the updater will identify affected unit proofs, assess their validity, and apply necessary updates. An empirical study will examine how software changes impact proofs, what invalidates them, and how they are fixed. To validate updates, the updater will reverify affected functions, comparing results with prior data using metrics like coverage and violation reports. Initially, the *proof repairer* (§III-3) will be used to resolve invalidated proofs. In future iterations, we plan to develop an LLM agent that, informed by empirical insights, can propose specific updates to the proof that realigns it with the changed software.

## IV. CONCLUSION

This poster outlines a research agenda for a unit proofing framework to support software verification and vulnerability discovery. Motivated by an assessment of existing unit proofs, the framework will provide tools and methods to develop, repair, and maintain unit proofs.

*Attribution*: An extended version of this abstract, with details of our result, was published in the 2025 International Conference on Software Engineering (NIER track) [16].

## REFERENCES

- [1] Dharun Anandayuvraj, Matthew Campbell, Arav Tewari, and James C Davis. FAIL: Analyzing software failures from the news using LLMs. In *the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE '24*.
- [2] Shweta Sharma. Counting the cost of CrowdStrike: the bug that bit billions. <https://www.cio.com/article/3478068/counting-the-cost-of-crowdstrike-the-bug-that-bit-billions.html>. Accessed: 2024-10-07.
- [3] M. Pistoia, S. Chandra, S. J. Fink, and E. Yahav. A survey of static analysis methods for identifying security vulnerabilities in software systems. *IBM Systems Journal*, 46(2):265–288, 2007.
- [4] Paschal C. Amusuo, Kyle A. Robinson, Tanmay Singla, Huiyun Peng, Aravind Machiry, Santiago Torres-Arias, Laurent Simon, and James C. Davis. ZTDS\_{JAVA}: Mitigating software supply chain vulnerabilities via zero-trust dependencies.
- [5] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.
- [6] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41(4):1–36, 2009.
- [7] Daniel Kroening and Michael Tautschnig. CBMC – c bounded model checker. In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 389–391. Springer, 2014.
- [8] Getting started - the kani rust verifier. <https://model-checking.github.io/kani/>. Accessed: 2024-10-01.
- [9] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. Testing or formal verification: DO-178c alternatives and industrial experience. *IEEE Software*, 30(3):50–57, 2013. Conference Name: IEEE Software.
- [10] Nathan Chong, Byron Cook, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz-Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle. Code-level model checking in the software development workflow. In *the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP '20*, pages 11–20.
- [11] Tong Wu, Shale Xiong, Edoardo Manino, et al. Verifying components of arm(r) confidential computing architecture with ESBMC. <https://arxiv.org/abs/2406.04375v1>. 2024. Accessed: 2024-10-04.
- [12] Ghada Bahig and Amr El-Kadi. Formal verification of automotive design in compliance with ISO 26262 design verification guidelines. *IEEE Access*, 5:4505–4516, 2017.
- [13] Kelechi G. Kalu, Taylor R. Schorlemmer, Sophie Chen, Kyle A. Robinson, Erik Kocinare, and James C. Davis. Reflecting on the use of the policy-process-product theory in empirical software engineering. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023*, pages 2112–2116.
- [14] Dirk Beyer and Thomas Lemberger. Software verification: Testing vs. model checking. In *Hardware and Software: Verification and Testing*. Springer, 2017. 99–114.
- [15] FreeRTOS/FreeRTOS/test/CBMC at main · FreeRTOS/FreeRTOS. <https://github.com/FreeRTOS/FreeRTOS/tree/main/FreeRTOS/Test/CBMC>. Accessed: 2024-10-08.
- [16] Paschal C Amusuo, Parth V Patil, Owen Cochell, Taylor Le Lievre, and James C Davis. A unit proofing framework for code-level verification: A research agenda. In *2025 47th IEEE/ACM International Conference on Software Engineering (New Ideas and Emerging Results track)*, 2025.
- [17] Mark Staples, Rafal Kolanski, Gerwin Klein, Corey Lewis, June Andronick, Toby Murray, Ross Jeffery, and Len Bass. Formal specifications better than function points for code sizing. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1257–1260.
- [18] Daniel Matchuk, Toby Murray, June Andronick, Ross Jeffery, Gerwin Klein, and Mark Staples. Empirical study towards a leading indicator for cost of formal software verification. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 722–732. ISSN: 1558-1225.
- [19] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Păsăreanu. Learning assumptions for compositional verification. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346. Springer, 2003.
- [20] D. Giannakopoulou, C.S. Pasareanu, and H. Barringer. Assumption generation for software component verification. In *Proceedings 17th IEEE International Conference on Automated Software Engineering*, pages 3–12, 2002. ISSN: 1938-4300.
- [21] Jamieson M. Cobleigh, George S. Avrunin, and Lori A. Clarke. Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. *ACM Trans. Softw. Eng. Methodol.*, 17(2):7:1–7:52, 2008.
- [22] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, and Houman Homayoun. Large language models for code analysis: Do LLMs really do their job? In *Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24)*, 2024.
- [23] Daye Nam, Andrew Macvean, Vincent Hellendoorn, Bogdan Vasilescu, and Brad Myers. Using an LLM to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ICSE '24*, pages 1–13, 2024.



## Manual Proofs Miss Critical Vulnerabilities

### Engineers need automated methods for developing and maintaining unit proofs

### What is Unit Proofing?

```

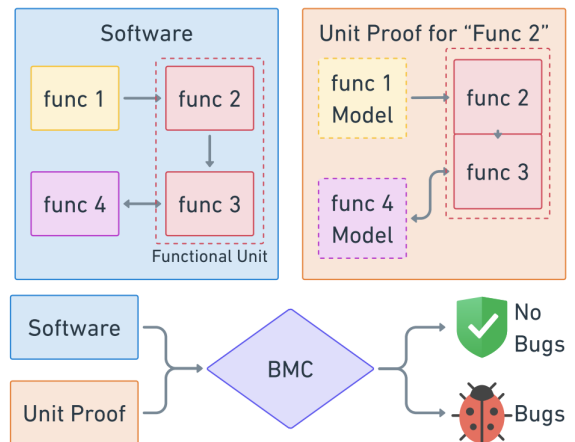
1  struct context {
2      uint8_t payload[CONSTANT];
3  };
4
5  int targetFunc(char *data, int len)
6  {
7      context *ctx =
8          get_current_ctx();
9      for (i=0; i<3; i++) {
10         ...
11     }
12     memcpy(ctx->payload, data, len);
13     _assert(sizeof(ctx->payload) >= len);
14 }

```

```

1  context *get_current_ctx() {
2      context *ctx =
3          malloc(sizeof(context));
4      _assume(ctx != NULL);
5      return ctx;
6  }
7
8  _unwind(targetFunc.0:4);
9
10 void harness() {
11     int len;
12     _assume(len <= CONSTANT);
13     char *data = malloc(len);
14     _assume(data != NULL);
15     targetFunc(data, len);
16 }

```



A software program with a potential security vulnerability.

A unit proof for verifying the software program.

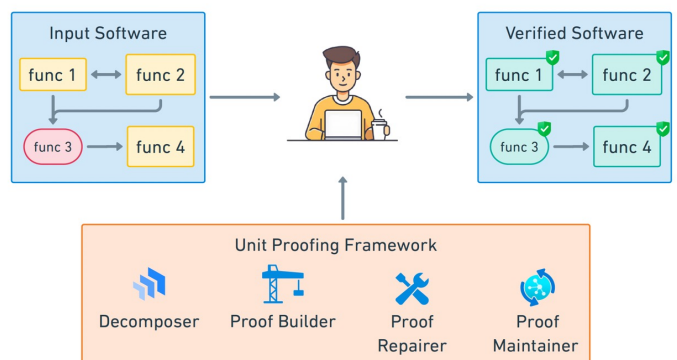
**Unit Proofing:** Software is decomposed, and unit proofs are created to verify units.

### Do manual proofs find vulnerabilities?

Recreated Vuln	Exposed	Reason	UP Stage
CVE-2018-16523	No	Insuff. properties	Design
CVE-2018-16524	No	Function Modeling	Development
CVE-2018-16525	No	Out-of-date Proofs	Maintenance
CVE-2018-16527	No	Insuff. Input Model	Development
CVE-2018-16600	No	Insuff. Input Model	Development
CVE-2018-16603	No	Insuff. Input Model	Development
CVE-2021-31571	Yes	—	—
CVE-2018-16526	Yes	—	—
CVE-2018-16599	Yes	—	—
CVE-2018-16601	Yes	—	—
CVE-2018-16602	Yes	—	—

Only 5/11 of CVEs were exposed. The remaining 6 were not exposed due to unit proof errors.

### Proposed Unit Proofing Framework



**Proposal:** Tools to assist software decomposition, unit proof creation, repair and maintenance.

Some organizations already use unit proofs.  
Let's make it easier for others !



Published in  
ICSE-NIER '25