SYSTEMATIC AND SCALABLE MEMORY SAFETY ASSURANCE FOR EMBEDDED SOFTWARE SYSTEMS.

by

Paschal Chukwuebuka Amusuo

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Elmore Family School of Electrical and Computer Engineering

West Lafayette, Indiana

August 2026

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. James Davis, Chair

School of Electrical and Computer Engineering

Dr. Aravind Machiry

School of Electrical and Computer Engineering

Dr. Benjamin Delaware

School of Computer Science

Dr. Saurabh Bagchi

School of Electrical and Computer Engineering

Approved by:

Dr. Milind Kulkarni

This dissertation is dedicated to the Almighty God for his blessings on my life and family.

ACKNOWLEDGMENTS

I begin by acknowledging the Almighty God, whose grace, mercy, and faithfulness have sustained me throughout this fulfilling journey. His divine guidance, strength, and wisdom have been my constant anchor.

I also appreciate my family for their constant support, prayers, encouragement, and sacrifices throughout this journey. Their belief in me has been a constant source of motivation.

I express my profound gratitude to my thesis advisor, Dr. James Davis, whose unwavering guidance, insightful feedback, and relentless support were instrumental, both in shaping this work and in my professional growth as a researcher. I am grateful for the opportunities he gave me and for his unwavering confidence in my abilities. His expertise, guidance and encouragement were invaluable throughout this journey.

I also extend my heartfelt thanks to my committee members, Dr. Aravind Machiry, Dr. Benjamin Delaware, and Dr. Saurabh Bagchi, for their insightful suggestions and constructive criticism during the course of my PhD program, which significantly strengthened the quality of this research.

Finally, I want to thank my friends and colleagues for their support and camaraderie during the many hours spent working on this research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

ABSTRACT

Memory-safety vulnerabilities remain a persistent source of software security and reliability failures. This risk is acute in embedded software, where systems commonly use C and C++, operate in critical environments, and may be difficult to patch. Existing techniques force a tradeoff between practicality and assurance: static analysis and fuzzing are useful for bug finding but provide limited assurance about the absence of memory safety errors, while formal verification provides stronger guarantees but often requires specialized expertise and costly manual modeling. This dissertation shows that systematizing and automating existing validation techniques can make stronger memory-safety assurance more achievable in embedded software practice.

This dissertation makes three contributions. First, it presents EmNetTest, a systematic testing framework that addresses the non-determinism and limited assurance of conventional dynamic analysis for embedded network stacks. EmNetTest mutates packet fields and explores protocol states to expose memory-safety defects. Across three embedded network stacks, it replicated 12 known vulnerabilities and discovered 7 new vulnerabilities in under three hours per target; same-seed random fuzzing found none within 24 hours. Second, it reduces formal-verification costs through a systematic unit-proofing process for component-level bounded model checking. The process constructs proof harnesses bottom-up using verification feedback, structured error-resolution steps, and explicit completeness criteria. Across four embedded components, systematic unit proofs exposed 74% of studied vulnerabilities without prior semantic knowledge, another 9% after parameter adjustments, and 19 new defects; the median proof verified 185 lines of C code and was developed in 72 minutes. Third, it automates proof-harness construction using principled techniques for selecting verification fidelity choices and a neuro-symbolic system that combines program analysis and large language models. Across four embedded operating systems, AutoSOUP verified memory safety for 93% of candidate targets, exposed 66.7% of evaluated CVEs, and identified 20 new vulnerabilities. Together, these contributions narrow the gap between practical bug finding and high-assurance verification for embedded software.

1. INTRODUCTION

1.1 Context and Problem Statement

Memory-safety vulnerabilities remain one of the most persistent and damaging classes of software defects. They occur when a program accesses memory outside the bounds, lifetime, or initialization conditions intended by the developer, as in buffer overflows, out-of-bounds accesses, and use-after-free errors. Such defects can crash systems, leak sensitive information, or enable arbitrary code execution. They account for up to 70% of vulnerabilities in memory-unsafe codebases [1, 2] and 75% of reported in-the-wild zero-day exploitations [3]. The Heartbleed vulnerability (CVE-2014-0160), for example, showed how a single bounds-checking error could be exploited to disclose secrets from process memory [4]. Even when they are not directly exploited, software defects can still cause severe operational failures, as illustrated by the July 2024 CrowdStrike outage [5–7].

These risks have led government, industry, and academia to place increasing emphasis on memory safety as a software engineering priority. In 2023, the United States Cybersecurity and Infrastructure Security Agency (CISA) publicly called attention to the urgent need for memory-safe software development practices [8]. DARPA has similarly supported initiatives aimed at advancing formal methods and other high-assurance approaches for cybersecurity resilience. Industry and academic leaders have likewise called for stronger memory-safety practices and standards in software development [9, 10]. Taken together, these efforts reflect a broad recognition that software systems increasingly require validation methods that are both practical to apply and capable of providing strong assurance.

This need is especially acute in embedded software systems. Embedded systems are widely deployed in safety-, mission-, and security-critical environments [11], are commonly implemented in C and C++ for performance, legacy, and toolchain reasons [12, 13], and are increasingly connected for communication, monitoring, and remote management. As a result, memory-safety vulnerabilities in embedded software can have serious security and operational consequences, and they have been the focus of repeated cybersecurity incidents and disclosures [14–16]. The problem is further compounded by the fact that embedded systems

often provide less isolation and fewer deployable hardening mechanisms than general-purpose platforms [17, 18], and may be costly or operationally difficult to patch once deployed [18].

Despite the importance of the problem, existing approaches for addressing memory-safety vulnerabilities force an unsatisfactory tradeoff between practicality and assurance. Techniques such as *static analysis* [19, 20] and *fuzzing* [21, 22] can be effective and practical for bug finding, but they are inherently incomplete: successful runs do not establish that memory-safety defects are absent. *Hardware-based mitigations* can strengthen defenses, but they are often impractical in constrained embedded settings [17, 18], and even when deployed, they may still be bypassed [23, 24]. *Formal methods* can provide much stronger, mathematically grounded guarantees for the properties they verify [25, 26], but their adoption in practice remains limited by the specialized effort and cost required to apply them effectively. This dissertation is motivated by the need to reduce this gap between practicality and assurance when validating memory safety.

It investigates how key validation tasks can be made more systematic and automated, so that stronger memory-safety guarantees become more achievable in embedded software practice. The central thesis is that techniques that systematize vulnerability discovery and automate memory-safety verification can advance a more practical, high-assurance approach to validating memory safety in embedded software systems.

1.2 Thesis

Memory-safety defects remain a critical and prevalent threat to software reliability and security, eliciting the need for higher-assurance validation methods. However, existing methods for validating memory safety either provide limited guarantees of memory safety (program analysis) or require costly and specialized expertise (formal verification). This problem is especially critical in embedded software systems where existing memory-safety validation methods are challenging to apply and errors are more costly to address after deployment. This dissertation addresses this gap by developing new systematic and automated techniques for high-assurance memory safety validation that do not require specialized expertise to adopt.

1.3 Scientific and Engineering Contributions

The contributions of this dissertation are two-fold: scientific and engineering.

1.3.1 Scientific Contributions

This dissertation contributes three techniques that reduce the gap between practicality and assurance in validating memory safety for embedded software systems. Together, they show that stronger memory-safety guarantees can be made more achievable by reducing dependence on non-deterministic dynamic testing, ad hoc, and manual proof-harness development.

Chapter 3: Systematic dynamic testing for embedded network stacks. This chapter addresses the non-determinism and limited assurance of conventional dynamic analysis for embedded network stacks. It contributes EmNetTest, a systematic testing framework that uses vulnerability-aware, field-level packet mutation and exhaustive exploration of reachable protocol states to expose packet-processing defects in embedded network stacks. The key insight behind this work is that vulnerabilities in embedded network stacks require only a few number of mutated fields in otherwise well-formed packets and protocol states to be exposed. The results show that systematic testing can make dynamic vulnerability discovery more predictable and effective, replicating known vulnerabilities and uncovering new defects in widely used embedded network stacks.

Chapter 4: Systematic unit proofing for component-level bounded model checking. This chapter addresses the high cost and impracticality of formal memory safety verification by reducing the need for ad hoc workflows and specialized expertise. It contributes a systematic process for performing component-level bounded model checking where proof harness development is guided by verification feedback, structured error-resolution steps, and explicit completeness criteria. The results show that bounded model checking through systematic unit proofing provides practical, high-assurance validation of memory safety, does not require high-fidelity bounds and models, and is cheap enough to be adopted in regular embedded software engineering workflows.

Chapter 5: Automated proof-harness generation for component-level memory-safety verification. This chapter advances the practicality of high-assurance memory-safety verification by automating the construction of proof harnesses. It formalizes the level of verification fidelity necessary for exposing memory-safety vulnerabilities, proposes principled techniques for selecting the design choices that influence verification fidelity (verification scope, bounds and environment models), and introduces a neuro-symbolic system to automate the construction of defensible proof harnesses that encode these choices. The results show that proof-harnesses generated using this method are of higher quality compared to those generated by existing techniques, and that they can expose security vulnerabilities during memory safety verification. This makes component-level bounded model checking more practical for real-world embedded software.

1.3.2 Engineering Contributions and Applications

Beyond its scientific contributions, this dissertation makes two primary engineering contributions.

1. **Publicly Available Tools for Embedded Software Validation:** This dissertation contributes publicly available software tools that engineering teams can use to validate memory safety in embedded software systems. *EmNetTest* [27] enables the systematic testing of TCP/IP stacks in embedded software and exposes packet-processing vulnerabilities through structured packet mutation and protocol-state exploration. *AutoSOUP* [28] enables the automatic generation of reusable proof harnesses for component-level bounded model checking in embedded software, reducing the manual effort required to verify memory safety in practice. Together, these tools provide practical support for high-assurance memory-safety validation in embedded software and can be integrated into earlier stages of the software development lifecycle.

2. **Discovery and Correction of High-Impact Cybersecurity Vulnerabilities:** The projects in this dissertation have also led to the identification and correction of high-severity vulnerabilities in widely used embedded operating systems and network stacks, demonstrating the real-world security value of systematic and high-assurance techniques

for memory-safety validation. The affected software includes projects such as FreeRTOS and Zephyr RTOS. These systems are among the most widely used open-source embedded operating systems, have been extensively tested using standard validation methods, and are maintained by prominent industry organizations, AWS and the Linux Foundation, respectively. Because these systems serve as foundational software for many downstream embedded applications and devices, improving their security can yield benefits that extend across a broad range of dependent systems.

Table 1.1. Affected products and CVEs resulting from the engineering contributions in this dissertation.

Chapter	**Affected product**	**CVE**
Chapter 3	Contiki-NG	CVE-2023-34100, CVE-2023-34101
		CVE-2023-37459, CVE-2023-37281
Chapter 3	PicoTCP	CVE-2023-35847, CVE-2023-35846
		CVE-2023-35849, CVE-2023-35848
Chapter 4	FreeRTOS	CVE-2024-38373, CVE-2025-5688
Chapter 4	Zephyr RTOS	CVE-2025-1675, CVE-2025-1674
		CVE-2025-1673, CVE-2025-9558, CVE-2025-9557
Chapter 4	Contiki-NG	CVE-2024-41126, CVE-2024-41125
Chapter 5	RIOT OS	CVE-2026-25139

Together, these engineering contributions provide concrete evidence that the scientific advances in this dissertation translate into practical tools and measurable real-world security impact.

2. BACKGROUND

2.1 Outline

This chapter provides the technical foundations that underpins the dissertation. It covers memory safety, embedded software, and techniques for validating memory safety. We defer concepts related only to specific chapters to the chapters where they are used.

2.2 Memory Safety

Prior work has defined memory safety in multiple ways. A common practical view defines it in terms of the absence of specific classes of errors, often referred to as *memory safety errors* or *memory corruption bugs* [23, 29, 30]. Other work defines memory safety more fundamentally as a semantic property of program execution, focusing on whether each memory access is valid according to the language and execution model [31–33]. In this section, I first discuss the major classes of memory safety errors and their consequences, and then present the definition of memory safety adopted in this dissertation.

2.2.1 Memory Safety Errors

Memory safety errors have been a longstanding concern in software engineering and computer security [23, 34]. They are software defects that cause a program to access memory in ways that are not valid for the referenced object. More specifically, they arise when a program reads from or writes to memory that has not been validly allocated, is outside the bounds of the intended object, or is no longer within the object's valid lifetime. These errors are particularly prevalent in low-level languages such as C and C++, where the programmer is responsible for managing memory explicitly and the language provides limited built-in protection against invalid memory accesses [13, 29].

Prior work commonly groups memory safety errors into two broad categories: *spatial* and *temporal* safety errors [29, 31].

- **Spatial safety errors** occur when a program accesses a memory location outside the bounds of the intended object, or otherwise accesses a location that is not a valid target

for the pointer. Common examples include out-of-bounds reads, out-of-bounds writes, and null-pointer dereferences. Such errors can corrupt adjacent memory, expose sensitive data, or cause the program to crash [4, 29].

- **Temporal safety errors** occur when a program accesses memory in a way that violates the valid lifetime of the referenced object. Common examples include use-after-free and double-free errors. These errors are dangerous because the affected memory may already have been reclaimed or reallocated for another purpose, allowing stale pointers to corrupt unrelated program state or influence future execution [23, 29].

These errors are both prevalent and consequential. A large body of prior work has shown that attackers can exploit memory safety errors to alter program behavior, leak sensitive information, trigger denial-of-service conditions, or gain control of the affected system [4, 23, 35]. Industry reports have further shown that memory safety errors account for a substantial fraction of vulnerabilities in C and C++ software and remain a dominant source of serious security exploits [1–3]. Their impact is not limited to security. Memory safety errors can also cause major reliability failures, including crashes and silent data corruption. For example, expert analysis disclosed in the 2013 Bookout v. Toyota trial identified multiple sources of memory corruption in Toyota's electronic throttle control software and linked task-death failures to unintended acceleration behavior [36]. More recently, the July 19, 2024 CrowdStrike outage, which disrupted millions of Windows systems worldwide and caused widespread operational and economic harm, was traced to an out-of-bounds memory read in the CrowdStrike Falcon sensor's content interpreter [5, 7, 37].

2.2.2 Definition of Memory Safety

This dissertation adopts the property-based definition from Nagarakatte [31] that defines memory safety as the property that all memory accesses performed by a program are well-defined according to the language specification.

This definition provides a precise basis for reasoning about memory safety assurance. Each memory access in the program must satisfy the conditions required for that access to be valid, such as referring to an allocated object, remaining within the object's bounds, and

respecting the object's lifetime. Similarly, these validity requirements can also be expressed as safety properties that must hold at each program point where memory is accessed, enabling both identification of violations during program execution and the formal verification of these program properties.

Under this formulation, memory safety errors can be understood as concrete violations of the underlying safety conditions governing memory accesses. Spatial errors violate properties about object identity and bounds, while temporal errors violate properties about object lifetime.

2.2.3 Memory Safety Assurance

This dissertation defines *memory safety assurance* as the measure of confidence that the memory safety properties of a system are always satisfied under realistic execution environments. This definition closely aligns with the defition of "secuity assurance" provided by the United States National Institute of Standards and Technology (NIST) [38].

According to NIST SP 800-39, the concept of assurance is closely related to the trustworthiness of an information system as it refers to concrete evidence that can be evaluated by third-party individuals or organizations to determine the extent to which the system can be trusted. In this context, memory safety assurance refers to the strength of the evidence provided about the memory safety of the target system. Such evidence can either be produced by design, through the use of security features or programming languages that enforces memory safety, or by validation, through the use of techniques that assess and provide evidence about the validity of memory safety properties of the system.

This dissertation therefore investigates techniques that provide stronger evidence of memory safety for embedded software while remaining practical and low-cost for use in real-world development workflows.

2.3 Embedded Systems and Software

Embedded systems are specialized computer systems that perform a dedicated function and are *embedded* within a larger mechanical or electronic system [12, 39]. They underpin

a broad range of physical devices and infrastructure, including industrial control systems, automotive systems, medical devices, and consumer electronics. We refer to the software components of these systems as embedded software. This embedded software typically monitors or controls the physical operations of the devices they are embedded within. Consequently, failures in these systems, whether caused by accidental or malicious actors, can impact physical processes and have severe consequences, including physical harm, economic disruption, and loss of life [11, 13].

Embedded software remains heavily exposed to memory-safety risk for several reasons. First, many embedded systems are implemented in memory-unsafe languages (C and C++) because of performance requirements, legacy codebases, hardware-level programming needs, and toolchain compatibility, where memory-safety errors are prevalent [12, 13]. Second, embedded software often processes data from untrusted sources, such as networks, peripherals, user commands, and external environments [39, 40]. Hence, vulnerabilities within them can be exploited by malicious actors to compromise the underlying system. Third, they are deployed in external environments with long service lifetimes where the cost of updating software is high. As a result, software defects can persist for years even after they have been discovered, causing long-term harm. Together, these factors make memory safety an especially important concern for embedded software systems [17, 18] and demonstrate the need for stronger assurances of memory safety in this software.

The importance of this problem is further amplified in foundational embedded software that play a critical role and are reused across many distinct products and deployments [12]. For example, embedded operating systems like FreeRTOS [41], Zephyr [42], and VxWorks [43] provide core functionalities for many embedded applications, including scheduling, memory management, inter-process communication, and drivers for peripheral devices. Similarly, embedded network stacks like lwIP [44] and Contiki-NG [45] provide communication and networking capabilities so that physical devices and infrastructure systems can be monitored and controlled remotely. These core software components are developed by a smaller set of vendors and relied upon by a broad ecosystem of downstream products and device manufacturers. As a result, a vulnerability in such software may therefore affect millions of downstream devices and systems [14, 16].

2.4 Validating Memory Safety

A broad range of techniques has been developed to validate the memory safety of software. These techniques differ mainly in two respects: the level of *assurance* they provide and the degree of *practicality* with which they can be applied.

In this context, *assurance* refers to the strength and scope of the evidence a technique provides about memory safety. In particular, it concerns what the technique can establish, under what assumptions, and what classes of violations may still go undetected. *Practicality* refers to the cost of applying the technique in real development settings, including tooling availability, computational overhead, required expertise, and compatibility with existing workflows, especially for embedded software.

This subsection reviews four major classes of techniques through this lens.

2.4.1 Static Analysis

Static analysis reasons about program behavior without executing the program. For memory safety, static analysis tools are commonly used to detect patterns associated with potentially unsafe memory accesses such as unchecked pointer dereferences, out-of-bounds accesses, use-after-free patterns, and unsafe dataflow into memory-accessing operations [20, 46–48].

Assurance. Static analysis can be effective at identifying likely memory-safety defects, but it usually provides limited assurance that a program is memory-safe as a whole. Most static analysis techniques rely on predefined rules, bug patterns, or abstract program models which trade precision or completeness for efficiency, which can cause real defects to be missed. As a result, they generally do not establish that all relevant memory-safety properties hold throughout the program [20, 46, 48].

Practicality. Static analysis is highly practical. Many static analysis tools are automated, scale to large codebases, and integrate easily into development workflows. This makes them attractive for routine use in software engineering practice.

2.4.2 Dynamic Analysis

Dynamic analysis evaluates software by executing it with concrete inputs and observing its runtime behavior. For memory safety, a prominent form of dynamic analysis is fuzzing [21], often combined with runtime memory-safety instrumentation such as AddressSanitizer to check the validity of memory accesses during execution and report violations when they occur [49]. These techniques have been highly successful in discovering memory-safety defects in practice [50–53].

Assurance. Dynamic analysis provides evidence only about the executions that are actually exercised. It can therefore confirm the presence of a memory-safety violation when one is observed, but it cannot establish that unobserved executions are safe. Its assurance is thus inherently limited: defects that are not triggered by the tested inputs or environments may remain undetected. Because dynamic analysis explores sampled executions rather than all possible behaviors, it does not provide general assurance that the program is memory-safe.

Practicality. Dynamic analysis is often practical and effective for bug finding as it does not require substantial manual effort or specialized expertise [21]. However, compared to static analysis, its practicality is more limited in embedded software settings. Embedded programs often depend on hardware peripherals, custom architectures, and timing-sensitive interactions that are difficult to model or reproduce faithfully in testing environments [54, 55]. These constraints can make dynamic analysis harder to deploy and can reduce the range of behaviors that they can realistically cover.

2.4.3 Formal Verification

Formal verification uses mathematical reasoning to determine whether a program satisfies a specified property [25, 26]. Unlike static and dynamic analysis techniques, which rely on abstractions or sampled executions, formal verification reasons about all behaviors captured by an explicit model of the program and its environment.

Assurance. Formal verification provides the strongest form of memory-safety assurance among these techniques. It can establish that specified memory-safety properties hold for all executions within the assumptions and bounds of the verification model. However, its

guarantees are only as strong as the assumptions, specifications, models, and bounds used during verification [56].

Practicality. Formal verification is usually the least practical to apply. Deductive verification often requires user-provided specifications, invariants, and proof guidance. Model checking can automate more of the reasoning, but it still requires formal environment models, verification harnesses, and explicit bounds on execution. Constructing these artifacts often requires deep system knowledge, substantial engineering effort, and specialized expertise. As a result, formal verification is often expensive to apply in real software development workflows [57, 58].

2.4.4 Memory-Safe Programming Languages

Memory-safe programming languages prevent broad classes of memory-safety errors. They achieve this through language design and compile-time enforcement. Rust, for example, uses ownership, borrowing, lifetimes, and type checking to prevent many spatial and temporal memory-safety violations, making it a promising option for systems and embedded software [13].

Assurance. Memory-safe languages provide strong default assurance for code governed by their safety mechanisms. However, this assurance may not extend to `unsafe` code, foreign-function interfaces, or libraries written in memory-unsafe languages, which still require separate validation [59].

Practicality. Memory-safe languages can reduce the need for separate memory-safety validation, but adopting them is often difficult for existing embedded systems. Many systems commonly depend on large C and C++ codebases, legacy libraries, and hardware interfaces that are costly or risky to rewrite. In addition, prior work [13] has also reported several challenges that impede Rust adoption for embedded software systems, such as the inadequacy of existing Rust software support and limited interoperability between Rust and C. As a result, memory-safe programming languages can complement other memory-safety validation methods, but cannot replace them entirely.

3. SYSTEMATIC DYNAMIC TESTING FOR MEMORY-SAFETY VALIDATION

3.1 Summary

Dynamic analysis is one of the most practical techniques for detecting memory-safety vulnerabilities in software systems. However, conventional fuzzing is often ineffective for embedded network stacks because these systems are highly stateful and require well-structured inputs to reach deep program behaviors. Its assurance is also limited, since validation depends on non-deterministic input generation and state exploration.

This chapter shows that dynamic testing can provide stronger guarantees of memory safety when it is made systematic. In particular, it demonstrates that systematic and vulnerability-informed input generation and state exploration algorithms make embedded network-stack validation more effective at exposing memory-safety vulnerabilities, increasing the strength of assurance provided while preserving the practical advantages of dynamic analysis.

Methodology: This work presents the first study of vulnerabilities in ENSs, focusing on their root causes and the packet inputs and protocol conditions that trigger them. Based on these findings, we design a dynamic testing approach that combines systematic protocol-state exploration with targeted field-level packet mutations and evaluate its effectiveness and time efficiency.

Findings: Our study shows that 77% of ENS vulnerabilities compromise memory safety, commonly due to missing validation of header fields or packet sizes. We find that 95% of these vulnerabilities can be triggered using valid packets with only one or two malformed fields or truncated sizes. Additionally, up to 30% require specific protocol states.

These observations inform a testing strategy that first drives the ENS into relevant protocol states and then mutates individual fields or truncates packets using interpolated values. Our prototype, EmNetTest, discovered 12 known vulnerabilities and 7 new ones across three ENSs, completing tests in under three hours per target. In contrast, fuzzing using random mutations and seeded with EmNetTest's inputs found no vulnerabilities within 24 hours.

Taken together, these results show that systematic automation can make dynamic memory-safety validation more practical and more reliable than conventional fuzzing, supporting the dissertation's thesis.

Statement of Attribution: The work in this chapter was published in the ASE 2023 proceedings [27]. I am the lead author of the paper and claim primary intellectual ownership of the research and content of this chapter.

3.2 Introduction

embedded network stacks (ENSs) are software components that enable network communication on embedded systems. There are several ENSs with varied architectures tailored to the semantics of individual embedded operating systems, such as Contiki-ng [45] and FreeRTOS [60]. Unlike the network stacks used by regular operating systems, ENSs run on embedded systems with limited or no vulnerability protections [17]. As a result, vulnerabilities in ENSs are severe and could be remotely exploitable. In the last five years, many critical defects have been discovered and reported in these ENSs [14–16]. Detecting cybersecurity vulnerabilities in ENSs remains an important challenge for securing the Internet of Things.

Automated software testing techniques for network stacks use formal methods, static analysis, and dynamic analysis to detect vulnerabilities. Formal methods, *e.g.,* model checking, provide strong guarantees [61–66] but are costly to apply and maintain. Static analyses efficiently find defects [19, 20, 67–69], but must be tuned to defect patterns and generate false positives. Dynamic analysis is promising, especially fuzzing [50–53, 70]. But while fuzzing ensures no false positives, it offers limited guarantees. No dynamic works examine the systematic testing of ENSs and consequently provide guarantees.

Our goal was to develop a systematic dynamic testing technique, one that could provide certain guarantees about the security of the ENS under test. But what guarantees should be prioritized? Several studies [71–73] show that defect patterns recur in software. Thus, identifying the characteristic defects can help prevent such defects in the future. We analyzed 61 security defects that were previously reported across 6 embedded network stacks to understand defect patterns. We found that most ENSs vulnerabilities occur because an ENS

directly used certain fields in packet headers without proper validation. Invalid values of such fields lead to out-of-bound (OOB) reads, buffer overflows, and integer wraparounds. We call these **packet validation vulnerabilities**. Our study also revealed that the test suites used in ENSs are inadequate to detect this recurring class of defect. To dynamically detect packet validation vulnerabilities, an approach must be systematic in varying fields (many different packet fields were problematic), able to reach different protocol states (many different protocol states were problematic), and able to find memory errors (most vulnerabilities involve OOB memory access).

Based on this analysis, we propose ZTD$_{\text{JAVA}}$, an automated and systematic framework for dynamic testing of ENSs. ZTD$_{\text{JAVA}}$ possesses three characteristics that enable it to uncover known vulnerability patterns in ENSs. *(1) Systematic packet generation:* ZTD$_{\text{JAVA}}$ systematically generates validly constructed packets with invalid header fields or truncated headers. *(2) Stateful:* ZTD$_{\text{JAVA}}$ provides sequences of packets that get the ENS to different protocol states before packet injection. *(3) Memory focused:* ZTD$_{\text{JAVA}}$ uses address sanitizers with dynamic memory poisoning to detect all memory corruptions. We implemented ZTD$_{\text{JAVA}}$ using PacketDrill which provides necessary scripting support for testing network stacks. We enhanced PacketDrill to support mutating arbitrary network packets.

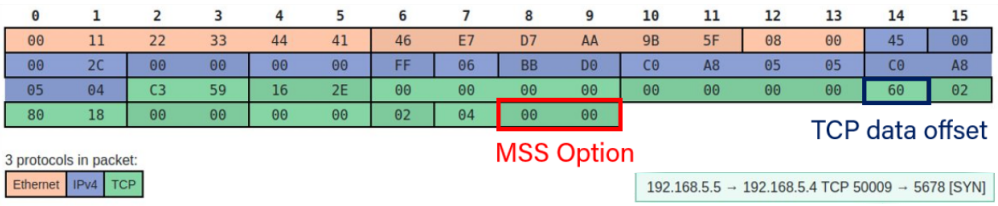

Figure 3.1. A hex representation of a TCP packet showing the TCP header length field and the TCP MSS Option Value. Listing 3.1 describes two CVEs associated with these fields in the FreeRTOS ENS.

We evaluated ZTD$_{\text{JAVA}}$ on 4 of the 6 ENSs whose vulnerabilities we studied: FreeRTOS, Contiki-ng, lwIP, and PicoTCP. We also created ENSBench, a dataset of 12 vulnerabilities by re-introducing previously known vulnerabilities into recent versions of the ENSs. Our evaluation showed that ZTD$_{\text{JAVA}}$ replicated all the 12 vulnerabilities we attempted. In

addition, ZTD$_{\text{JAVA}}$ found 7 new vulnerabilities (zero days), which can be remotely exploited by any user and potentially allow arbitrary code execution. We compared our framework with fuzzing. We ran 4 fuzzers from the Poncelet *et al.* benchmarks [70] on the latest version of Contiki-ng (which contains 5 vulnerabilities) and found that within 24 hours, no fuzzer detected any of the vulnerabilities.

Our work shows the importance and effectiveness of systematic testing for detecting critical software defects. We invite the community to explore systematic testing approaches, beyond the current trend of automated randomized testing (fuzzing).

In summary, we contribute:

1. We perform the first comprehensive study (§3.6) of 61 reported ENS vulnerabilities, understand their root causes, and provide insights into the packet sequences that trigger these vulnerabilities.

2. We designed and implemented ZTD$_{\text{JAVA}}$ (§3.7), an automated systematic testing framework for ENS. Our evaluation shows that ZTD$_{\text{JAVA}}$ effectively finds known and new vulnerabilities in ENS.

3. As part of our framework, we implemented PacketDrill++, an extended version of PacketDrill that facilitates adversarial testing of network stacks and can be used independently of our testing framework.

4. ENSBench: A dataset of 12 recreated and 7 new vulnerabilities, packaged into recent versions of ENSs to support the evaluation of other defect detection tools. ZTD$_{\text{JAVA}}$ detects all vulnerabilities in this dataset.

3.3 Background

3.3.1 Embedded Network Stacks (ENS)

embedded network stacks (ENSs) enable network connectivity for embedded systems. ENSs are either part of an embedded operating system (Integrated ENS) [60, 74] or standalone libraries (Standalone ENS) [44, 75]. ENSs follow a layered software architecture, each layer implementing a specific protocol on the TCP/IP stack.

Listing 3.1. CVE-2018-16523 and CVE-2018-16524: Snippet showing a divide-by-zero defect triggered by the TCP MSS Option (green) and an out-of-bound read (blue) triggered by the TCP Data Offset. Both are in the FreeRTOS network stack. ZTD_{JAVA} can recreate these vulnerabilities (Table 3.7).

```
1  static void prvCheckOptions(...) {
2   const unsigned char *pucPtr = ... ;
3   const unsigned char *pucLast = pucPtr +
4   (((pxTCPHeader->ucTCPOffset >> 4) - 5) << 2);
5   while(pucPtr < pucLast){
6    ...
7    else if(( pucPtr[0]  == TCP_OPT_MSS) &&
8      (pucPtr[1] == TCP_OPT_MSS_LEN)) {
9     uxNewMSS = usChar2u16(pucPtr + 2);
10    if(pxSocket->u.xTCP.usInitMSS > uxNewMSS){
11     ...
12     pxTCPWindow->xSize.
13     ulRxWindowLength = ((uint32_t) uxNewMSS) *
14     (pxTCPWindow->xSize.ulRxWindowLength /
15     ((uint32_t) uxNewMSS ));
16        ...
17   }}
18   pucPtr += ...
19   ...
20 }}
```

ENSs vulnerabilities pose a greater threat than those of regular network stacks due to the absence of operating systems and hardware vulnerability protection mechanisms. Regular operating systems, *e.g.,* Linux, provide more protection in their OS design. This includes features such as Address Space Layout Randomization (ASLR), Data Execution Prevention (DEP), address space isolation, and Stack Canaries [76] that prevent the exploitation of memory vulnerabilities. Also, modern processors include no-execute (Nx) regions that prevent the unauthorized execution of codes in sensitive memory regions [77]. Many embedded OSes and processors lack these features [17, 18], increasing the ease of vulnerability exploitation.

ENSs are designed for embedded systems, which are resource-constrained, have real-time requirements, and often lack common library support. ENSs are also tailored to the underlying embedded operating system's threading and scheduling semantics. Consequently, they differ from regular operating systems' network stacks, which use the POSIX standard [78] for portability. For example, the `accept` call in FreeRTOS blocks until a successful TCP connection is established or the timeout elapses. The `accept` call in lwIP is non-blocking and defines a callback that would be called on successful connection establishment. Meanwhile, Contiki-ng has no `accept` syscall. Instead, it uses an event-driven callback for all events, including a Socket connection event.

3.3.2 Internet Protocols and Network Packets

In this work, we focus on Internet Protocol (IP) or TCP/IP suite, which includes various protocols that specify how data should be packaged, addressed, and routed [79, 80]. The TCP/IP suite is organized into a layered architecture (Figure 3.2). An Internet packet has elements for each layer of this architecture, recursively structured as headers associated with one layer and a payload associated with the next (Figure 3.1). The protocol's implementation processes the corresponding headers at each level and passes the payload along.

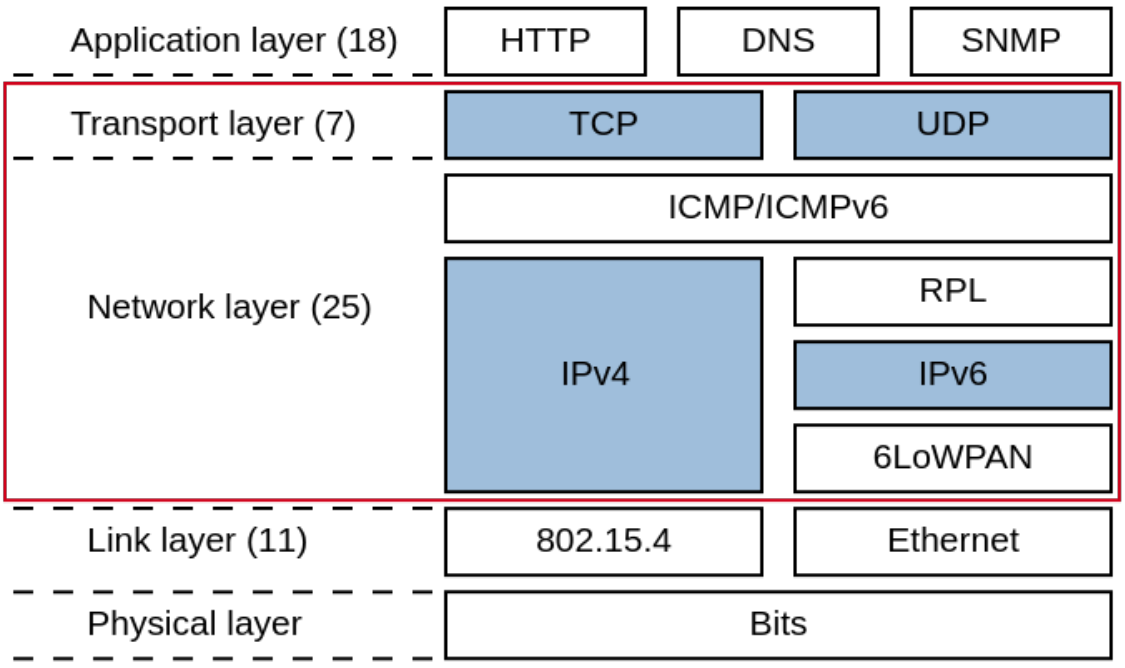

Figure 3.2. Layers of the TCP/IP stack. Our tool targets layers in the red box (specific protocols in blue). Values in parentheses indicate number of analyzed CVEs in each layer (§3.6.2).

3.3.3 ENS Vulnerabilities

ENSs are usually implemented in C/C++ for performance and compatibility reasons. They handle complex packet structures across multiple layers. Hence, ENSs are prone to defects that can be cybersecurity vulnerabilities. With the absence of sufficient protection mechanisms, the vulnerabilities in an ENS can be exploited to either disable or remotely control the entire system. Furthermore, these vulnerabilities can be triggered remotely by any use with network access to the system.

Listing 3.1 shows the snippet corresponding to two vulnerabilities, CVE-2018-16523 and CVE-2018-16524. CVE-2018-16523 (divide by zero) occurs because the TCP MSS option value, `uxNewMSS`, is used as the divisor to calculate `RxWindowLength`. A TCP packet with an MSS value of zero will lead to a divide-by-zero error. CVE-2018-16524 (out-of-bounds read) occurs because offset in the header, *i.e.,* `ucTCPOffset` is used to compute a pointer

address `pucLast`, which is later read through `pucPtr`. These vulnerabilities are triggerable remotely without authorization by sending TCP packets. Furthermore, these vulnerabilities can be exploited to gain control of the system because of the lack of isolation mechanisms in embedded systems.

3.4 Related Work

3.4.1 Traditional Testing

Many ENSs incorporate test suites that help the maintainers validate the various functionalities they develop. As shown in §3.6, these test suites are inadequate. Although automated test generation tools [81–83] exist, the tests generated by them are inadequate at finding faults [84]. Furthermore, domain knowledge is required to use these automated test-generation tools effectively.

Research and commercial tools exist to facilitate the easy development of test suites for network stacks. PacketDrill [85], a network stack testing tool that enables the use of scripts to test the end-to-end correctness behavior of network stacks. PacketDrill focuses on testing functionality and always generates valid packets, *i.e.,* has valid and well-formed headers. However, as found in §3.6, most vulnerabilities occur because of invalid values in packet headers. InterWorking Labs has commercial testing solutions for testing network protocols and also uses malformed packets [86]. However, access costs over $10,000,[1] limiting its adoption in open-source projects and the low-margin embedded systems marketplace [87].

3.4.2 Formal Methods

Several tools [64–66] have explored formal methods for verifying network functions. Zaostrovnykh [64] and Pirelli [66] developed formal verification tools to automatically prove that a network function conforms to a provided specification. However, these techniques do not apply to multithreaded programs such as ENS. Microsoft's Project Everest [88] verifies various components of HTTPs and has provided verified implementations of some cryptographic libraries. FreeRTOS, maintained by AWS, also verifies their network stack

[1] ↑This quote was provided to us through personal communication.

implementation, FreeRTOS+TCP [89]. These formal methods are used to verify specific correctness properties of the network protocol implementations and do not make complete guarantees about their security. As shown by Fonseca *et al.* [56], formal methods guarantees are only as good as their underlying assumptions. Hence, we still need to assess the security of these systems through dynamic testing.

3.4.3 Fuzzing

Fuzzing [21] has found many software defects. From a network perspective, fuzzing has been mainly explored to find bugs in network applications. AFLNet [50], StateAFL [51], and SnapFuzz [52] are three recent works in this direction. These works focus on setting up a proper communication channel with a network application and sending test data to the application through well-formed network packets. A recent work, TCPFuzz [53] uses fuzzing and differential testing to detect semantic vulnerabilities in the transport layer. TCPFuzz always generates valid packets and cannot find vulnerabilities arising from invalid packets.

Poncelet *et al.* [70] applied several state-of-the-art fuzzing tools to test individual functions of the Contiki-ng ENS. They reported that testing lower-layer functions does not get deep penetration. Conversely, directly testing upper network layers increases the rate of false positives as some inputs and corresponding packets are impossible as lower layers will reject them. Furthermore, they fail to trigger code paths that require the network stack to be in a particular state. This is demonstrated in our evaluation (§3.8) where ZTD_{JAVA} found various vulnerabilities in the well-test portions of Contiki-ng.

3.4.4 Vulnerability Studies

Several researchers have studied vulnerabilities' characteristics in different software systems [90–94]. Most of these works focus on well-provisioned systems, *e.g.,* desktop and web software. Few works study vulnerabilities in embedded systems. Al-Boghdady *et al.* [93] studied the characteristics of security vulnerabilities in IoT operating systems. While they focused on characterizing the CWEs (Common Weakness Enumeration) reported by static analysis tools, their study does not cover how these vulnerabilities are triggered or detected.

Similar to our work, Malik & Pastore examined CVEs in Edge frameworks and found that (1) the network components were a common source of CVEs, and (2) specific values were often problematic, but did not go into detail on ENSs nor evaluate a solution [95]. Other industry practitioners have also published reports of security vulnerability analyses of ENSs they conducted. For example, Zimperium [96] published a blog post containing details of the vulnerabilities they discovered in FreeRTOS, and Forescout published a report containing an analysis of the 33 vulnerabilities they found and a list of observed common anti-patterns [15]. No prior work systematically analyzes vulnerabilities in ENSs.

3.5 Knowledge Gaps and Research Questions

This work aims to fill two gaps. First, no study characterizes cybersecurity vulnerabilities in ENSs. Second, no dynamic system exists to systematically detect ENS cybersecurity vulnerabilities. We ask:

Theme 1: Vulnerability analysis

RQ1: What are the types and root causes of vulnerabilities?

RQ2: What packet sequences trigger ENS vulnerabilities?

Theme 2: State of practice for packet validation testing

RQ3: Are ENS tested for packet validation vulnerabilities?

Theme 3: Evaluating systematic testing with EmNetTest

RQ4: To what extent can bounded systematic testing uncover packet validation vulnerabilities?

3.6 ENS Vulnerabilities and Testing (RQ1-3)

This section presents methodology and results for RQ1-3. To summarize our findings, ENS CVEs are typically *packet validation vulnerabilities*. The studied ENS incorrectly handle packets that are slightly malformed, sometimes from a particular protocol state. In 95% of CVEs, 1-2 fields are incorrect. Repairs often involve a single `if`-statement.

Table 3.1. Embedded network stacks (integrated and standalone) whose CVEs we examined. C/C++ LoC (source, not tests) measured with cloc[97]; for integrated ENS we measured only the network implementation. GitHub data as of May 2023.

Name	**Size (LOC)**	**GitHub stars**	**GitHub forks**	**# CVEs studied**	**# CVEs recreated**	**# new vulns.**
FreeRTOS(+TCP)	42.2K	3.6K (76)	1.2k (110)	11	5/5	0
Contiki-ng	41.6K	1.1K	635	24	2/2	2
Zephyr	95.7K	7.7K	4.8K	11	*Not attempted*	*Not attempted*
PicoTCP	32.7K	1K	201	12	5/7	4
LwIP	84.3K	525	249	1	*Not attempted*	1
FNET	18.0K	106	46	2	*Not attempted*	*Not attempted*

3.6.1 Methodology

Repository Selection

We studied both integrated ENS and standalone ENS (Table 3.1). We selected ENSs integrated into major open-source embedded operating systems. From lists in survey papers [12, 98], we selected three embedded OSes with over 1K GitHub stars: Zephyr (maintained by Linux Foundation), Contiki-ng (Supported by Swedish Research Institute), and FreeRTOS (maintained by AWS). From a previous vulnerability study [16], we selected the top-3 actively-maintained repositories (by GitHub stars) with reported CVEs. These were PicoTCP, LwIP, and FNet.

Data Collection

We obtained vulnerability reports (CVEs) from the National Vulnerability Database (NVD) [99]. We searched the NVD for the associated project. For integrated ENSs, we only considered vulnerabilities in the networking stack. There were 81 total CVEs. We discarded 15 CVEs that omitted technical vulnerability details. After preliminary analysis, we observed 61 of the remaining 66 vulnerabilities were caused by the poor validation of packets received by ENS. We termed these *packet validation (PV) vulnerabilities*. We removed the 5 non-PV vulnerabilities.

Data Analysis

One author analyzed each vulnerability report and technical details, including screenshots explaining vulnerable code, links to the vulnerability's GitHub issue, and the repairing pull request (PR).[2] We indicate the specific extracted features below — these are a typical set of features in software failure analysis [100]. For soundness, a second author analyzed a random sample of 13 vulnerabilities. We measured interrater agreement using Cohen's Kappa score [101]. We obtained κ=0.82, indicating substantial agreement [101].

3.6.2 RQ1: Vulnerability Characteristics

Finding 1: Memory Out-of-Bound Read and Write are the most common vulnerabilities (70%).

Finding 2: Missing length field validation and Missing packet size validation are the most frequent root causes and account for 69% of vulnerabilities.

Finding 3: The network layer contains most vulnerabilities (41%), followed by the application layer (29%). Vulnerabilities are also found in every layer of the stack.

We describe CVE types, root causes, and affected components.

[2]↑During this analysis, we found 3 new CVEs (excluded from our analysis).

Vulnerability Types

First, we group CVEs according to their Common Weakness Enumeration (CWE) [102].

Table 3.2. Proportion of CVE types. "Others": double-free, DNS cache poisoning, division-by-zero, and infinite loops.

Type	**# CVEs (%)**
Out-of-Bounds Read (CWE 125,126,200)	22 (36%)
Out-of-Bounds Write (CWE 120,121,122,787)	21 (34%)
Integer Overflow (CWE 191)	5 (8%)
Integer Underflow (CWE 190)	4 (7%)
Null-pointer dereference (CWE 476)	4 (7%)
Other	5 (8%)
Total	61 (100%)

Table 3.2 shows the result by this taxonomy. Memory over-read/write (the first two rows) comprise 70% of the vulnerabilities.

Implementation Root Causes

We studied code and repairs to learn the implementation-level root causes of CVEs. Table 3.3 groups these into several recurring patterns. Roughly 69% of CVEs in ENSs (first two rows) result from missing checks on length fields and data packet size.

Table 3.3. Implementation-level root causes of CVEs.

Root cause	# CVEs (%)
Missing length field validation	23 (38%)
Missing packet size validation	19 (31%)
Missing header value validation	7 (12%)
Missing integer wraparound validation	2 (3%)
Other	10 (16%)
Total	61 (100%)

Two examples:

- **Missing length field validation (CVE-2018-16524):** FreeRTOS uses the TCP header length field to calculate the size of the TCP options region. However, it fails to validate the length value. Consequently, an invalid length value results in arbitrary memory read.
- **Missing packet size validation (CVE-2022-36054):** Contiki-ng receives a 6LoWPAN packet and after header compression, copies the packet into a buffer. If the packet is the first fragment of a fragmented packet, only 148 bytes are allocated. Contiki-ng doesn't verify the received packet size before copying it into this buffer. Consequently, a buffer overflow could result in a remote code execution attack.

Vulnerable Layers

The left column of Figure 3.2 shows the distribution of CVEs across the ENS layers. The top layers for CVEs are network (41%) and application (29%).

3.6.3 RQ2: Packet Sequences That Trigger CVEs

Finding 4: 95% of vulnerabilities depend on one or two fields and consequently can be triggered with a maximum of two field changes. 40 different fields contribute to these vulnerabilities.

Finding 5: 30% of CVEs are stateful, *e.g.,* involving an existing connection or a specific protocol state.

Here, we study packet sequences that can trigger these CVEs. Each packet sequence has a prefix (*i.e.,* state prefix) $p_1 p_2 \dots p_{k-1}$ that brings the ENS to a vulnerable state, followed by the vulnerability-triggering packet p_k. For instance, consider a vulnerability in processing a TCP FIN packet. To trigger the vulnerability, we first need to send packets that can set the ENS to a state where it accepts a FIN packet. Then, we send a FIN packet triggering the vulnerability. Understanding both parts enables a testing scheme to uncover real CVEs.

Properties of the Vulnerability-Triggering Packets (p_k)

Here, we investigate two aspects: (1) *Root Cause Fields* (RC_f): Which incorrectly-handled fields result in vulnerabilities? (2) *Dependent Fields* (D_f): How many fields of a packet does a vulnerability depend on?

For instance, consider CVE-2018-16599, which is caused by the incorrect validation of the UDP header length field. The vulnerability can be triggered only if the NBNS Type field is NET_BIOS (0x0020) and the NBNS Flags field indicates a response packet (0x8000). Here, $RC_f = 1$ (for the length field), whereas $D_f = 3$ (for the length, type, and flags fields).

Table 3.4 shows RC_f and the number of CVEs resulting from it. We see that 57 (93%) of CVEs arise from fields in the protocol headers and options that are incorrectly handled. Table 3.5 shows the distribution of CVEs according to D_f. Most vulnerabilities (58, or 95%) have $D_f \leq 2$. However, it is not just one field that is problematic — 40 different fields across 15 protocols contribute to the 61 CVEs.

Table 3.4. Distribution of CVEs based on the incorrect fields (RC_f) in the CVE-triggering packet. These fields often included those specifying the length of the packet or option component (rows 1-2), or specific values of other fields or options (rows 3-4). Often, the packet was truncated (row 5).

Type	Count(%)
Header length value	8 (13%)
Option length value	8 (13%)
Header field value	24 (39%)
Option value	2 (3%)
Truncated packet	15 (25%)
Others	4 (7%)
Total	61 (100%)

Table 3.5. Distribution of CVEs by # of dependent fields (D_f).

# Dependent Fields	# CVEs
1	34 (56%)
2	24 (39%)
> 2	3 (5%)
Total	61 (100%)

Properties of the Packet Sequence Prefix

We studied the vulnerable code and execution path to identify any states involved. Table 3.6 shows that 70% CVEs are stateless (can be triggered with a single packet/no prefix) and that the remaining 30% (12 CVEs) depend on the state of the system. Of these, 13 CVEs, occurring on stateful protocols, require the protocol to be in a specific set of states. 5 other CVEs depend on the properties of the previously-sent packet(s).

Table 3.6. Distribution of vulnerabilities based on the statefulness required to expose the vulnerability.

State Required	Protocol	# CVEs
Stateless	–	43 (70%)
Requires protocol state	TCP	6 (10%)
Requires protocol state	RPL	1 (2%
Requires protocol state	BLE	2 (3%)
Requires protocol state	MQTT	4 (7%)
Requires packet sequence	6LoWPAN	3 (5%)
Requires packet sequence	802.15.4	2 (3%)
Total	All	61 (100%)

3.6.4 RQ3: Test-Suite Characteristics

We analyzed the test suites of four ENSs to understand why the known CVEs, which we discussed in §3.6.2, existed. Based on the CVE characteristics, we looked for four aspects of validation: (1) Unit tests involving input packet processing operations with malformed input; (2) Capability of injecting specific (and possibly malformed) packets; (3) Tests involving packets with invalid headers (cf. Table 3.4); and (4) Tests involving statefulness (cf. Table 3.6).

Finding 6: ENSs are validated using end-to-end simulation tests and unit tests. The actual implementations of these tests are unique in each ENS (no standard test framework).

Finding 7: While some ENSs include packet injection tests, these are regression tests for specific CVEs. One ENS provides packet seeds and a harness for stateful fuzzing.

Finding 8: None of the ENSs systematically check invalid header or option fields, nor include unit tests for the various operations performed on an input packet.

FreeRTOS: FreeRTOS validates its ENS with end-to-end tests, unit tests, and formal verification. The end-to-end tests use the sockets interface to establish network connections and validate behaviors of the network stack. They provide (incomplete) memory safety

proofs for main packet processing functions. Not all functions are verified and the provided proofs depend on the corrections of some unverified functions. FreeRTOS has some packet injection tests with invalid headers, but all cases are regressions for past CVEs.

Contiki-ng: Contiki-ng is validated with network simulation using cooja [103], a packet injection test, and fuzzing. The network simulation tests involve various end-to-end tests under different simulated network environments. Their packet injection tests use a fixed set of network packets. These are mostly regression tests to check for previous defects or vulnerabilities. Their packet injection framework is also used for fuzzing.

PicoTCP: PicoTCP validates with unit tests and end-to-end demo applications. The provided unit tests are mostly on non-packet related tasks, such as IP address-to-string conversion and socket tests. The demo applications test supported protocols in different network environments.

LwIP: LwIP validates with unit tests, network stress testing, and fuzzing. Their unit tests mostly test the output operations of the network stack, not the input packet processing functions. Several unit tests configure the test socket to specific protocol states. For stress, they measure the reliability of simulated networks while increasing the number of nodes and messages in the network. They also provide a fuzzing harness and fuzzing seeds for the different protocols.

3.7 EmNetTest: Design and Implementation

3.7.1 Design Requirements

Based on our findings from Theme 1, an automated testing framework to detect PV vulnerabilities in ENSs should have three characteristics:

- **Ability to Detect Memory Issues:** Based on Finding 1, it should detect memory corruption vulnerabilities.
- **Systematic Packet Generation:** Based on Findings 2 and 4, it should systematically generate valid test packets with incorrect header values and truncated headers.
- **Stateful:** Based on Findings 3 and 5, it should drive the ENS stack to different protocol states for multiple protocols.

Such a framework would improve the state of the art in ENS testing (cf. §3.4 and Findings 6-8).

3.7.2 Design

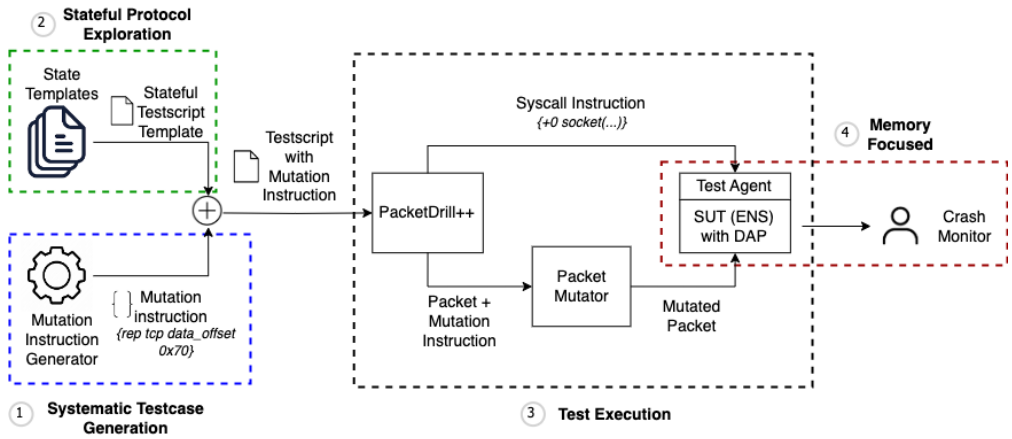

Figure 3.3. Overview of the design of EmNetTest. It systematically generates mutation instructions by repeatedly taking a protocol header, selecting combinations of fields, and iterating through possible values (blue box ①). It achieves statefulness by using a set of test script templates that can explore different protocol states (green box ②). Packetdrill++ interprets each test script and sends the syscall and packets to the SUT (ENS). The per-ENS Test Agent maps received POSIX syscall instructions to appropriate behavior and execute the behavior on the ENS (black box ③). The SUT is instrumented with dynamic poisoning and compiled with ASAN to aid the detection of memory corruption (red box ④).

EmNetTest meets these requirements. Figure 3.3 illustrates.

- **Memory focused:** Address Sanitization (ASAN) [49] with ENS specific instrumentation.
- **Systematic packet generation:** An ordered generation algorithm can systematically generate all packets but is prioritized for packets that trigger known PV CVEs.
- **Stateful:** We build on the PacketDrill tool [85]. It provides packet sequences that cover some relevant protocols. We extend it to additional protocols and employ a seed set of state-covering sequences (packet sequence prefixes).

Listing 3.2. Systematic packet generation. The caller imposes order by working from smaller to larger N and varying S.

```
# Input: Num. entities N, stride S, packet p_k
# Output: Yields next packet for this config.

# SELECTION of N fields and options
{f_1, ···, o_1, ···} = nextPacketEntities(N) # Generator

# INTERPOLATION
vals_{f_1} = interpolate(f_1, S)
...
vals_{o_1} = interpolate(o_1, S)
...

# GENERATION (uses Python itertools)
for f_1 in vals_{f_1}:
  ...
    for o_1 in vals_{o_1}:
      ...
        p_k.modify(f_1, ···, o_i, ···)
          yield p_k
# Caller sends prefix + |p_k| and checks result
```

Custom Address Sanitization with Dynamic Address Poisoning (DAP)

Memory corruptions (*i.e.,* out-of-bounds read and write) in embedded systems may not lead to program crashes [104] (`SEGSEGV`). ASAN is a well-known technique to convert memory corruptions into program crashes. However, ASAN assumes the target application is using standard memory allocation and deallocation functions (*e.g.,* `malloc`/`free`) — which is not the case with ENSs, as they use custom allocators. To handle this, we use ASAN's Dynamic Address Poisoning (DAP) support. For each ENS, we modified its custom allocators such that after every allocation, the corresponding memory chunk will be unpoisoned (*i.e.,* OK to use). Similarly, we modify deallocator or release functions to poison the corresponding memory chunk (*i.e.,* Invalid to use).

We modify packer copying routines to detect out-of-bound memory accesses during packet processing. Specifically, after a packet is received and copied into the buffer, we poison the rest of the allocated buffer that is not covered by the received packet. Then, ASAN will detect any bytes read or written beyond the bounds of the allocated buffer.

(Ordered) Systematic Packet Generation

We systematically generate ordered test packets. *Systematic* means all packets are generated. *Ordered* means an ordering over the packets such that likely-useful packets are generated early.

We describe our approach, formalized in Listing 3.2. We assume a prefix sequence of packets $p_1 p_2 \ldots p_{k-1}$ to reach a desired protocol state, followed by test packet p_k (§3.6.3). The algorithm generates all valuations of p_k.

Systematic: Packet p_k is a sequence of bytes consisting of required header fields, optional fields, and a payload. The payload was not the cause of ENS CVEs (§3.6) so we exclude it. Different subsets of the header and option fields are selected to modify (generator on line 5). We obtain possible values for each following an interpolation sequence from minimum (*e.g.,* `0x00`) to maximum (*e.g.,* `0xff`) along a stride S (line 7). All combinations are explored (line 13). This ensures we can detect vulnerabilities that either depend on the minima or maxima,

or on a range of values as determined by the stride. To generate all p_k, choose maximum N and a Stride of 1.

Ordered: We order the generation of p_k starting from 0 entities (the original p_k), then 1 entity, and so on, discarding repeating packets. This order places the likely-to-be-useful packets early in the sequence — per Table 3.5, most CVEs depend on at most 2 fields (*i.e.,* $D_f \leq 2$). This suggests that most of the vulnerabilities could be found with $N = 2$. During ENS validation, engineers may parameterize by bounding the maximum number of fields to select N. They may trade exhaustiveness vs. cost via the search stride S.

Handling truncate: As reported in Table 3.4, 25% of the analyzed CVEs involved a packet that was truncated to shorter than the expected length. The caller of Listing 3.2 generates these with modest post-processing: remove bytes from the end of the packet and then update checksums in earlier layers.

Stateful

In our CVE study, we found that many CVEs can only be triggered from certain states of a protocol. We examined existing network testing tools to identify one that can reach many states of a protocol. We chose the PacketDrill tool. It is designed to drive a network stack through the state machine for various protocols [85]. It supported two transport-layer protocols (TCP, UDP) and two network-layer protocols (IPv4, IPv6), with a corpus of >200 scripts that test different functionalities of the TCP protocol.

We developed a corpus of 7 Packetdrill scripts that can reach the 7 different TCP states where a packet can be injected (`LISTEN`, `SYN-SENT`, `ESTABLISHED`, `FIN-WAIT-1`, `FIN-WAIT-2`, `LAST-ACK`, `CLOSE-WAIT`). We had only one UDP script as UDP has no states. We use these scripts as test script templates. As shown in Figure 3.3, for each test case and mutation instruction generated, we append the mutation instruction to all template scripts of the protocol being tested. This enables us to test all protocol states with the same input.

3.7.3 Implementation

Our EmNetTest implementation is 3,426 lines of C/C++ and Python. We describe pertinent aspects of the implementation.

Portability

The purpose of EmNetTest is to support many ENSs. Portability is a priority. As noted in §3.3, ENS have diverse architecture and semantics. We de-coupled the EmNetTest packet generation from the delivery and evaluation of packets. PacketDrill generates a combination of socket commands and network packets. As shown in Figure 3.3, a per-ENS Test Agent maps POSIX socket commands to the appropriate behavior on the ENSs. This includes both minor naming changes (*e.g.,* socket vs FreeRTOS_socket) as well as more substantial semantic changes (*e.g.,* rendering the asynchronous socket semantics of LwIP into synchronous POSIX semantics). This test agent (server) on the ENS receives socket interactions and packets and delivers them to the ENS. The rest of the system is agnostic to the ENS under test.

PacketDrill++

We implemented the network testing tool as an extension of PacketDrill. We extended the PacketDrill grammar to support mutation instructions. We implemented a packet mutator component in C that, given a packet and a set of mutation instructions, mutates the packet following the instructions. For example, the instructions might be to change the value of a field, insert an option, and truncate the packet. We modified PacketDrill so that it loads the Packet mutator as a shared library and uses it for packet mutation. PacketDrill++ also interacts with our portable bridge component instead of directly with the network.

Packet Injection

EmNetTest uses a virtual network interface (TAP [105]) to send mutated packets to Embedded Network Stack. A virtual network interface allows us to inject packets at the lowest layer of the network stack, which simulates the exact same behavior as when the ENS receives the packet from the internet, removing the possibility of false positives. Furthermore, a virtual network interface does not introduce the same network latency that would be introduced by a normal network interface connected to the internet.

Parallelizing Test Execution

Once packets are generated systematically, executing them is an embarrassingly parallel problem. We decoupled test case generation from execution using the producer-consumer pattern, saturating our servers.

Deduplicating Vulnerabilities

EmNetTest systematically generates packets, which may result in many redundant defects. Our crash monitor analyzes the observed failures and deduplicates them based on the stack trace (line of crash).

Linux Versions of ENSs

Although ENSs support many boards, they also support Linux as a development environment [106]. EmNetTest uses the Linux versions of the ENSs. This enables EmNetTest and the ENS to run on the same machine, enhancing communication between them. This does introduce the risk that our results mask defects in HW/SW integration on real boards, *e.g.,* due to layering issues [107].

3.8 RQ4: Systematic Testing with EmNetTest

We evaluate our systematic testing framework by running EmNetTest on 4 embedded network stacks. Our evaluation aims to understand the extent our systematic testing ap-

proach can uncover packet validation vulnerabilities. Specifically, we answer the following questions.

- **RQ4.1:** Can EmNetTest replicate known vulnerabilities?
- **RQ4.2:** Can EmNetTest discover new vulnerabilities?
- **RQ4.3:** What are EmNetTest's performance characteristics?
- **RQ4.4:** How does EmNetTest compare to fuzzing?

3.8.1 Experimental Setup

We evaluated N = 1,2,3 and we used a stride that yielded 4-6 values for each field (Listing 3.2).

We used the following servers for our experiments: two 32-core machines (Ubuntu 22.04, Intel Xeon W-2295 CPU@3GHz); and one 64-core machine (Ubuntu 22.04, AMD EPYC 7543P CPU@2.8GHz).

3.8.2 Embedded Network Stack Selection

We selected 4 ENSs for our evaluation — FreeRTOS+TCP, Contiki-ng, PicoTCP, and LWIP. These stacks from §3.6 had the highest proportion of Network and Transport layer vulnerabilities, suiting them for EmNetTest.

3.8.3 ENSBench: Vulnerability Dataset Construction

To enable us to answer RQ4.1, we replicated 12 known vulnerabilities in recent versions of 3 selected ENSs — FreeRTOS, Contiki-ng, and PicoTCP.[3] These were selected out of the 14 reported vulnerabilities that affected the IPv4, IPv6, TCP, and UDP protocols in the selected ENSs layer protocols. We skipped 2 vulnerabilities because Packetdrill lacked support for the features they required (IPv6 fragmentation). We studied their fixing commits to replicate the vulnerabilities and reverted the fix. Porting the vulnerabilities to the latest

[3]↑LwIP had no reported IP/TCP vulnerabilities.

version allowed us to have all vulnerabilities in a single build for testing. Table 3.7 describes the CVEs we recreated.

Table 3.7. CVEs EmNetTest recreates. The last column indicates dependent fields and kind of changes that expose CVE. Notation: F—set header **F**ield; O—insert+set **O**ption; T—Truncate header; Rd—**Rea**d; Wr—**Wr**ite.

ENS	**CVE-ID**	**Type**	**Operators**
FreeR-TOS	2018-16523	Div-by-zero	1 (O)
	2018-16524	OOB Read	1 (F)
	2018-16526	OOB Write	1 (O)
	2018-16601	Integer underflow	1 (F)
	2018-16603	OOB Read	1 (T)
Contiki-ng	2021-21281	OOB Read	1 (F)
	2022-36053	OOB Write	2 (F, T)
Pi-coTCP	2020-17441	OOB Read	2 (F, F)
	2020-17442	Integer Overflow	2 (F, O)
	2020-17444	Integer Overflow	2 (F, O)
	2020-17445	OOB Read	2 (F, O)
	2020-24337	Infinite Loop	1 (O)

3.8.4 RQ4.1: Replicating Known Vulnerabilities

To evaluate EmNetTest's ability to expose defects, we ran EmNetTest on vulnerable versions of FreeRTOS, Contiki-ng and PicoTCP. Our test found all vulnerabilities in the tested stacks as listed in Table 3.7 using a maximum of 2 mutations. Table 3.7 also shows the mutation types performed on the packet that exposed each vulnerability. These vulnerabilities

were triggered by a total of 9 distinct fields. IPv6 extension header length caused 3 while TCP data offset caused 2.

3.8.5 RQ4.2: Discovering New Vulnerabilities

To evaluate EmNetTest's ability to discover new defects, we ran EmNetTest on recent versions of the ENS listed in §3.8.3. For FreeRTOS and Contiki, we only ran the tests for IPv4 and IPv6 respectively as that was the only IP version they supported. Table 3.1 shows the count of vulnerabilities we found in each of the selected stacks. Table 3.8 describes the various vulnerabilities that we found. In our artifact, we also included the specific scripts that exposed each vulnerability and a detailed description and impact of each vulnerability.

Contiki-ng and PicoTCP confirmed the vulnerabilities we reported, assigned CVE identifiers, and repaired the vulnerabilities. We have been unable to establish communication with the LwIP team.

Table 3.8. New vulnerabilities EmNetTest found. Notation: Same as Table 3.7.

ENS	CVE ID	Description	Configuration
FreeRTOS	—	*No vuln found*	
Contiki-ng	2023-34100	OOB Rd (TCP MSS)	1 (O)
	2023-37459	OOB Rd (TCP flags)	1 (T)
PicoTCP	2023-35847	Div-by-zero (TCP MSS)	1 (O)
	2023-35846	OOB Rd (TCP fields)	1 (T)
	2023-35849	OOB Rd (IP checksum)	1 (F)
	2023-35848	OOB Rd (TCP MSS)	2 (O, O)
LwIP	L1	OOB Rd (TCP options)	1 (O)

We attempted to evaluate EmNetTest on commercial ENSs. We contacted five vendors of real-time OSes and embedded network stacks: WindRiver (VxWorks), Segger (emPower

OS, embOS, emNet), Green Hills Software (GHNet), Lynx (LynxOS), and Sysgo (PikeOS). All declined to allow us to evaluate on their systems.

3.8.6 RQ4.3: Performance Characteristics

Test Execution Duration: We measured the execution duration of EmNetTest by varying the number of dependent fields (*i.e.,* N in Listing 3.2). Table 3.9 shows the test execution duration on PicoTCP. For each value of N, we executed tests over all supported protocols (IPv4, IPv6, TCP, UDP) using 32 consumer instances. Our results in §3.8.4 and §3.8.5 show that all reported and new vulnerabilities could be found with only N=1 and N=2 tests.

Table 3.9. Performance results from testing on PicoTCP.

Task	# Test cases	Instances	Time
One test case	1	1	0.5 sec
N=1 test	2,211	32	2.13 min
N=2 test	134,296	32	2.21 hr
N=3 test	5,303,604	32	63.17 hr

Coverage Analysis: We analyzed the coverage achieved by running EmNetTest on 4 ENSs compiled with `gcov`.

Table 3.10. Table showing the line coverage achieved by different tests when executing EmNetTest on all the tested stacks.

Test	FreeRTOS	Contiki	PicoTCP	lwIP
Script 1	37.4%	25.4%	11.3%	32.6%
Script 2	34.1%	24.8%	5.9%	29.9%
All Scripts	51.3%	29.6%	12.7%	40.0%
N=1 tests	53.4%	33.4%	14.8%	43.6%

Table 3.10 shows the line coverage achieved executing different tests. For the Integrated ENSs, we consider only the coverage of the networking component. The first two rows show the coverage for two PacketDrill tests with scripts representing different TCP states. The third row represents the coverage achieved when we ran stateful test scripts representing all TCP states. The last row indicates the coverage when we ran a systematic test with N=1. By using test scripts that represent different TCP states, we achieved a significant increase in coverage. The little coverage increase caused by N=1 tests shows that packet validation vulnerabilities exist in codes that are covered by normal executions. As shown in Table 3.8, this N=1 was also sufficient in detecting most of the new vulnerabilities we found. We could not get very high coverage as the ENSs contained protocol implementations in other network layers that we don't currently support.

3.8.7 RQ4.4: Fuzzing Comparison

We used the Contiki-ng fuzzing benchmark provided by Poncelet *et al.* [108] to demonstrate that within a time budget, fuzzing is not deterministic in uncovering vulnerabilities. We selected 4 fuzzers from the benchmark (MOpt [109], Intriguer [110], SymCC [111], and AFL). The first 3 had the best results during Poncelet *et al.*'s evaluations. AFL is a standard comparison point. To help the fuzzers, we (1) disabled checksums in Contiki-ng, and (2) augmented the fuzzers' seed set with EmNetTest's comprehensive set of seed packets.

Table 3.11. Fuzzing results with the Contiki-ng fuzzing benchmarks after 24 hours. Version 1 contains a version of Contiki-ng used by the Poncelet *et al.* authors for evaluation. It contains the vulnerabilities reported by the authors in their paper. Version 2 is the most recent commit on Contiki-ng on GitHub as of May 1st, 2023. This version contains 5 vulnerabilities, including 2 detected by EmNetTest. These vulnerabilities should cause a crash in V2 if triggered.

Metrics	**Version 1 [70]**	**Version 2**
# paths covered	190	316
Crashes found (#)	21	0
Hangs found (#)	12	0

After 24 hours, the second column of Table 3.11 shows none of the fuzzers triggered any crash in vulnerable Version 2.

Comparison with Other Network Protocol Fuzzers: As noted in §3.4, there are other network fuzzers, *e.g.,* TCPFuzz [53] and AFLNet [50]. They are not appropriate for the vulnerabilities we studied. For example, AFLNet targets the application layer of the network stack, while TCPFuzz is concerned with semantic defects on legitimate input.

3.9 Discussion

EmNetTest vs. Fuzzing vs. Static Analysis: Fuzzing is the most used technique for detecting vulnerabilities. As a dynamic analysis technique, fuzzing provides a low rate of false positives. But fuzzing requires significant computing resources to be effective, limiting developers' ability to detect vulnerabilities at development time.

Like fuzzing, EmNetTest is also a dynamic analysis technique. But unlike fuzzing, it completes and provides guarantees that the known patterns of packet validation vulnerabilities do not exist in the ENS.

Static Analysis is another widely used approach that succeeds in detecting specific vulnerability patterns. Unlike dynamic analysis, many static analysis techniques consider only specific code sections rather than the entire software and give off a lot of false positives

[112]. We briefly ran CodeQL [113] on the ENSs repositories and found that it struggled with inter-procedural cases, failing to find any known or new vulnerabilities we detected.

Learning from Intra- and Interproduct Vulnerabilities: Our results in §3.8.5 show that the known vulnerability patterns still exist in ENSs. We found cases in Contiki-ng where they added regression tests for individual errors but failed to generalize these tests to classes of errors. We recommend that software engineers learn from the individual errors that occur in their software, and prepare generalized test cases that can detect similar errors.

We also found that the same vulnerabilities recur in different software implementations. For example, CVE-2023-35847 (§3.8.5) in PicoTCP is the same as CVE-2018-16523 in FreeRTOS. CVE-2023-34100 in Contiki-ng is the same as CVE-2018-16524 in FreeRTOS. Anandayuvaraj *et al.* already performed preliminary studies on this phenomenon of recurring failures in software engineering [114] and initiated conversations towards a failure-aware software development lifecycle [71, 115]. Our findings in this paper further emphasize this need for software engineers to learn from the reported vulnerabilities and failures of other software products. Furthermore, tools like EmNetTest can help by ensuring that new vulnerability patterns, discovered in one software product, can be easily detected and fixed in every other similar software product they may exist in.

Integrating Security Protections into Embedded Firmware: As shown in §3.6.2, memory corruption is the most prevalent class of vulnerability reported in ENSs. In addition to detecting and fixing these vulnerabilities, protection mechanisms could also be implemented to harden the embedded devices and mitigate the impact of exploitations. Protection techniques such as stack canaries, Address Space Layout Randomization (ASLR), and No-Execute (Nx) regions exist for regular operating systems which makes vulnerability exploitation difficult. Unfortunately, Yu *et al.* [17] showed that these security protection techniques are missing in embedded systems. Prior research [87, 116, 117] identified cost as a factor that limits the integration of security in embedded systems. Our work further illustrates the importance of integrating these security defenses into embedded systems. Hence, we advocate for further research in developing and deploying cost-effective protection mechanisms in embedded systems.

Improving Testing Practices for Open-Source Software: As seen in §3.6.4, different ENSs employ diverse methods and implementations for testing. While many of the test suites had unit tests, the size and robustness of the tests varied across different ENSs. This suggests the need for a standard test framework for testing similar software systems such as ENSs.

Future Work: We identify the following opportunities for research to improve this work.

- *Checkpoint-based Optimization:* Stateful testing involves driving the ENS to a specific state before injecting the test packet. This introduces significant overhead. The use of a deferred forkserver [118] does not work on multi-threaded or networked applications. In the future, we hope to explore the use of process checkpointing and recovery [119] to optimize the execution of stateful tests.
- *Smart Test Case Generation:* Our current EmNetTest design depends on generating packets where all combinations of fields, up to a value k, can be mutated at a time. We plan to explore using program and dataflow analysis to understand which packet fields interact or depend on each other during packet processing and prioritize testing the combinations of these interacting fields. This approach will build on existing research on concolic and hybrid testing that integrates static analysis, dynamic analysis, and symbolic execution to aid vulnerability detection [120, 121]. This improvement will lead to optimizations in the time to run multiple field combinations shown in §3.8.6.
- *Application to Other Protocols:* Our results show that vulnerabilities in all network protocol implementations share similar patterns. Hence, we have two questions. Would we find similar patterns in the implementations of other protocols? Would vulnerabilities in different implementations of the same protocols or protocol groups (*e.g.,* cryptographic protocols) share the same patterns?

3.10 Limitations and Threats to Validity

Limitations of EmNetTest: While EmNetTest is effective in discovering packet validation vulnerabilities, it has the following limitations

- Our implementation of EmNetTest is limited to the protocols supported by Packetdrill (TCP, UDP, IPv4, IPv6, and ICMP). We believe EmNetTest will work for protocols in other layers as they contain vulnerabilities with similar patterns.
- Due to the different architectures of ENSs, EmNetTest requires a distinct Test Agent for each ENSs. We designed a portability layer that makes it easy to implement a Test Agent for any ENS.

Construct Validity: We studied CVEs using well-known classifications, reducing threats to construct validity. To mitigate further, we used inter-rater agreement as a check.

Internal Validity: We assessed the testing practices of ENSs by looking at their test suites. The maintainers of these ENSs may have other testing processes which we don't know about.

External Validity: We mitigate one generalizability concern by examining multiple ENS of both kinds (integrated and standalone). We acknowledge that the CVEs in our study (§3.6) may have been found by a small number of persons using specific techniques. There may be other vulnerability patterns in ENSs not detected or reported. Nevertheless, the patterns we observed in these CVEs helped us find new vulnerabilities.

4. SYSTEMATIC UNIT PROOFING FOR MEMORY-SAFETY ASSURANCE

4.1 Summary

Component-level Bounded Model Checking can provide much stronger memory-safety assurance than testing or program analysis, but it is costly to adopt because proof harness construction depends on detailed understanding of program semantics, ad hoc workflows, and labor-intensive environment modeling.

This chapter demonstrates that proof harnesses sufficient to expose a majority of memory-safety vulnerabilities can be developed systematically, without prior semantic knowledge and without costly high-fidelity models. As a result, component-level bounded model checking can produce strong, auditable memory-safety evidence at much lower engineering cost.

Methodology: We introduce a bottom-up unit proofing process that begins with simple, unconstrained unit models and iteratively refines them using coverage metrics and verification errors, with objective criteria for model completeness and validation against the remaining software. We apply this process to 73 functional units across four embedded software components and evaluate vulnerability detection and proof-construction effort.

Findings: Our evaluation shows that 74% of memory-safety vulnerabilities were exposed using systematically developed unit proofs without prior semantic knowledge. An additional 9% were revealed by adjusting bounded model checking parameters, while the remainder were not exposed due to limitations of the underlying verification tool. By only verifying functions with known vulnerabilities, we found 19 new ones.

The median unit proof covered 185 lines of C code and was developed in 72 minutes, suggesting approximately 1200 lines per workday. In contrast, a prior method reported by AWS [122] achieved only 1500 lines per month, or 50 lines per workday. These results show that systematic proof development can make high-assurance memory-safety verification far more practical, directly supporting the dissertation's thesis.

Statement of Attribution: The work in this chapter was published in the ICSE 2026 proceedings [123]. I am the lead author of the paper and claim primary intellectual ownership of the research and content of this chapter.

4.2 Introduction

Memory safety defects [35] remain a significant threat to the reliability of critical software systems, particularly in programs written in C and C++. These defects facilitate cybersecurity attacks [3], software crashes [124] and outages [5]. A notable example is the July 2024 CrowdStrike outage [5], caused by an out-of-bounds read [6], affecting 8.5 million systems [125] and resulting in a $10 billion loss [7]. Additionally, these defects account for up to 70% of high-severity vulnerabilities [1, 2] in C/C++ codebases. While memory-safe languages like Rust exist, many active C/C++ projects remain, with barriers to transition, and many Rust projects still depend on C/C++ libraries and unsafe code [13, 126]. Given the severity of these issues, government agencies [8, 127], academia [10], and industry [9] increasingly advocate for memory-safe development practices and stricter safety standards. Particularly, the United States Department of Defense encourages the use of formal methods to mitigate these critical defects [127].

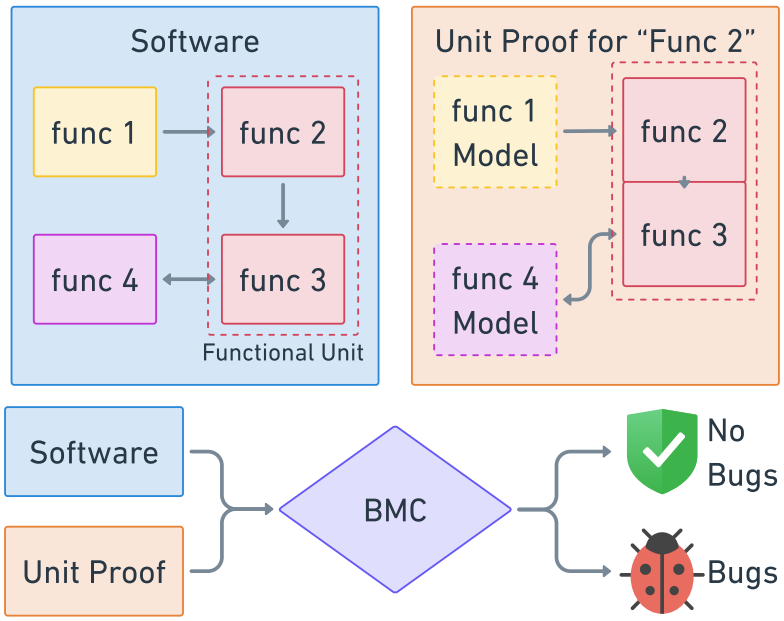

Figure 4.1. A unit proof, containing models for func1 and func4, verifies the functional unit comprising func2 and func3.

To mitigate memory safety defects, AWS [122], ARM [128], and Ant Group [129] employ Bounded Model Checking (BMC) to formally verify memory safety of software systems. They adopt a compositional approach, known as unit proofing [130], to verify functional units individually. This approach relies on *unit proofs* that correctly model a unit's environment for verification. However, there is no standardized method for creating unit proofs to ensure correctness. Different organizations employ varying methodologies, leading to inconsistencies even within the same projects. The lack of a systematic approach introduces errors in unit proofs (§4.6.4), prevents the detection of defects [130], and increases the cost of adoption for new organizations. In addition, unit proofing remains understudied in research. Existing experience reports [122, 128, 129] only provide high-level descriptions without empirical validation. Prior studies on formal verification [131–133] do not address unit proofing or memory safety verification. As a result, no systematic methods exist for developing unit proofs and there are no empirical evaluations of unit proofing's effectiveness or costs.

This work presents the first empirical study on unit proofing for memory safety verification and addresses five research questions. We evaluate on embedded software due to its safety-critical applications [11] and the challenges of adopting alternative techniques (§4.5.1).

First, we introduce a systematic method for unit proof creation, where coverage and error reports guide the unit proof development, and objective criteria are used to assess completion. Using this method, we created 73 unit proofs to verify functional units in four embedded operating systems. Using a blind approach (§4.5.3), we then reintroduce 89 known memory safety defects in these units and evaluate the effectiveness of unit proofs in detecting the recreated defects, analyze the unit proof characteristics and measure the cost of developing and using them. Finally, we compare them with expert-developed unit proofs and assess if the systematic approach will generalize to unverified functions.

Our results provide empirical measurements of the benefits and costs of using unit proofs for memory safety verification. Of 89 recreated defects, systematic unit proofs detected 66 (74%) and an additional 8 (9%) with increased BMC bounds, while 10 remained undetected due to memory exhaustion. Additionally, unit proofs exposed 19 new defects, highlighting their practical impact. On cost, developing each unit proof took an average of 87 minutes, with execution averaging 61 seconds. The median unit proof was developed in 72 minutes, executed in 25 seconds, and verified 185 lines of C code. Our results also show that unit proof sizes correlate with functional unit size ($R^2 = 0.328$); that verification formula size better predicts execution time than does program size ($R^2 = 0.852$ *vs.* $R^2 = 0.029$); and that systematic unit proofs are smaller, faster, and achieve better coverage than those developed by project experts.

In summary, our main contributions are:

- A systematic approach to developing unit proofs for memory safety, including strategies for deriving unit proof models and loop bounds.
- The first empirical study on the use of unit proofs for memory safety verification. Our results provide evidence that unit proofs are cost-effective for ensuring memory safety.
- A dataset containing 73 unit proofs, 89 recreated and 19 new memory safety defects to facilitate research on unit proofing.

Significance: As software has become more critical to society, formal methods are now seeing broader industry adoption. Our investigation aims for two benefits: (1) to help engineering teams to develop effective unit proofs, and (2) to guide research on automating

unit proof development, thereby reducing unit proofing costs and enabling broader adoption of memory safety verification in software engineering practices.

4.3 Background & Related Work

In this section, we cover the memory safety concept and how it can be verified using Bounded Model Checking (BMC). Then we discuss approaches for applying BMC to larger software and review the empirical studies on formal verification.

4.3.1 Memory Safety

The C and C++ languages allow software engineers to manipulate memory directly without restrictions. While this provides flexibility and performance benefits, it requires engineers to ensure *memory safety* — that no memory is accessed unless it has been explicitly allocated and with valid scope [23]. Violating this property poses significant risks, including denial of service, information leaks, and remote code execution [23, 35]. Such defects are the leading causes of security vulnerabilities in memory-unsafe codebases [1, 2] and are frequently exploited in zero-day attacks [3].

Szekeres *et al.* [23] describe various ways memory safety defects arise in practice. First, a memory reference is manipulated to point to an invalid memory location. This can result from untrusted input being directly used to compute the reference or from programming errors such as unchecked allocation failures and arithmetic errors that render the reference invalid. In the second step, this invalid reference is used to access memory, either for reading or writing. These steps may occur within the same function (*intra-function*, *e.g.*,, in func2 of Figure 4.1), across different functions in the same functional unit (*intra-unit*, *e.g.*,, in func2 and func3), or across distinct functional units (*inter-unit*, *e.g.*,, in func2 and func4).

Several techniques are commonly used to detect or mitigate these defects, including static analysis [46, 134], dynamic analysis [27, 135], and runtime mitigation [24, 136–138]. In this work, we provide empirical measurements of a complementary and emerging formal technique for mitigating memory safety defects.

Listing 4.1. The left program contains a potential memory safety defect and an expected memory safety property (in red). The right shows a unit proof used to verify the program. It contains preconditions that model input variables (in blue) and loop unrolling bounds (in purple).

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

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

4.3.2 Bounded Model Checking for Verifying Memory Safety

Bounded Model Checking (BMC) is a verification technique that mathematically checks whether a program, within specified *bounds*, satisfies a given property [139]. Properties can be *specific* to the program or *agnostic* (*e.g.,* memory safety properties). A program can be bounded by restricting the number of loop iterations, recursion depth, size of data structures or its access to children functions. To verify a property, BMC encodes the program and the specified property as a Boolean Satisfiability (SAT) problem and solves it using SAT/SMT solvers to identify any sequence of instructions within the program that can violate the specified property [140].

Various BMC tools, such as the C Bounded Model Checker (CBMC)[141] and Kani[142], automate memory safety verification by inserting assertions at memory access points (see red line in Listing 4.1) to verify the safety of these operations. These assertions are translated into properties and analyzed by constraint solvers. Additional tooling is also available to support engineers using BMC tools. For example, the CBMC viewer [143] converts CBMC

verification and coverage reports into browsable HTML and JSON formats. The CBMC Proof Debugger [144], a VS Code plugin, aids in debugging error traces. The Goto Harness [145] assists in generating initial unit proofs for specific functions. Our work provides an empirical study of compositional bounded model checking, an emerging technique for verifying memory safety.

4.3.3 Scaling BMC to Software of Realistic Sizes

Compositional BMC

While BMC is effective for small programs, Boolean satisfiability is NP-complete, which limits BMC's scalability [146]. Compositional (or modular) BMC [147] addresses this limitation by decomposing programs into smaller, verifiable units, verifying each unit individually, and deriving guarantees for the entire program compositionally. This can be achieved through two approaches [148]. Specification-based methods use predefined specifications to guide a unit's verification, allowing each unit to be verified independently. Conversely, fully automated approaches [149–152] decompose, verify, and integrate verification results without human intervention. While compositional BMC enables scaling BMC to larger software, it can potentially miss *inter-unit defects* where the memory reference invalidation and invalid memory access occur in different units (§4.3.1).

Unit Proofing

Unit proofing is a specification-based approach for compositional BMC where *unit proofs* (the specification) are developed and used to verify individual software units. Amusuo *et al.* [130] describe this engineering activity of developing, using and maintaining unit proofs as "unit proofing". Many organizations [122, 128, 129] have reported using this approach to verify the memory safety of critical software components. As shown in Figure 4.1, unit proofs serve as function-level harnesses, modeling the unit's environment (input variables, shared state, and undefined functions) and specifying necessary BMC bounds (loop bounds,

unit scope [1], data structure bound [2]). Listing 4.1 presents a sample unit proof for the `targetFunc` program, where blue lines model input variables, and the `get_current_ctx()` function models an actual implementation. Unit proofing is conceptually similar to unit testing, as both validate isolated software units. However, unlike unit tests that check correctness for specific inputs, unit proofs verify correctness for all inputs satisfying the provided models.

While multiple organizations have reported using unit proofs, their approach to creating the unit proofs vary: AWS [122] develops unit proofs from scratch, while ARM [128] and Ant Group [129] derive them from existing function specifications and unit tests, respectively. In all cases, unit proof creation was done in collaboration with development teams, with AWS iteratively refining models until they were accepted as accurate. Additionally, AWS and Ant Group reported increased bug detection rates due to unit proof adoption. However, none of these organizations disclosed detailed information on the models and BMC bounds used, how they derived them or the cost of developing the unit proofs.

4.3.4 Empirical Studies on BMC

Few studies have evaluated the effectiveness and cost of Bounded Model Checking (BMC). Unit proofing experience papers [122, 128, 129] report discovering new defects but lack a systematic validation of effectiveness. Beyer *et al.* [133] showed BMC outperforms fuzzing and testing in detecting defect on software benchmarks, but their study did not account for challenges in realistic software, where loop bounds [153] or incorrect models [130] can hinder detection. Thus, an empirical evaluation of BMC in real-world software or through unit proofing remains missing.

Regarding cost, few studies assess the cost of applying BMC to realistic software. AWS reported that a new engineer required one month to develop high-quality unit proofs and verify 1,500 lines of code [122]. Additionally, Huang *et al.* [58] found that most formal verification projects required over a year of effort and verification expertise. Given the huge

[1]↑A unit's scope is the source files compiled with the unit proof. Function definitions in these files are included during verification. Other functions remain undefined.
[2]↑bound on array and linked-list sizes.

cost of formal verification, empirically measuring unit proofing costs is therefore essential for assessing its practical adoption.

4.4 Research Questions

We summarize as two knowledge gaps

Gap 1: Lack of a Standardized Unit Proofing Approach: Reports [122, 128, 129] show organizations develop unit proofs using different methods, with inconsistencies even within the same project (§4.6.4). This increases error likelihood and compromises formal verification guarantees [130]. Additionally, the absence of a standardized approach raises adoption costs, as new organizations may struggle to identify effective or cost-efficient methodologies.

Gap 2: No Empirical Evaluations of Unit Proofing: There has been no empirical assessment of the effectiveness, cost, or limitations of unit proofing for memory safety verification. Existing studies on formal verification [131–133, 154] either do not cover memory safety verification or focus on benchmark programs which will not account for defects that span multiple functional units. Consequently, engineering teams lack data on the benefit-cost tradeoffs of unit proofing, hindering informed adoption decisions.

Given the variations in unit proofing approaches, their sometimes costly need for project expertise, and the absence of benefit-cost measurements, this work investigates the following question: *If we develop unit proofs using a uniform and systematic approach, would they expose memory safety defects? If so, what do they cost?* We divide this into five **research questions** across three themes:

Theme 1: Unit proof effectiveness

- **RQ1:** Does a systematic unit proofing approach enable detection of memory safety defects?

Theme 2: Unit proof characteristics and cost

- **RQ2:** What are the characteristics of the unit proofs produced?
- **RQ3:** How long does it take to develop and execute unit proofs?
- **RQ4:** How do unit proofs developed systematically differ from those developed by project experts?

Theme 3: Generalizability of approach

- **RQ5:** Will the systematic unit proofing approach generalize across other functions?

This paper presents a methodology that uses verification feedback to guide unit proof development and a set of objective criteria to assess completeness and quality. Theme 1 evaluates whether unit proofs developed with this approach expose memory safety defects. Theme 2 examines their characteristics and the cost of developing and using the unit proofs. Theme 3 investigates the generalizability of this approach beyond the studied sample.

4.5 Methodology

4.5.1 Study Design

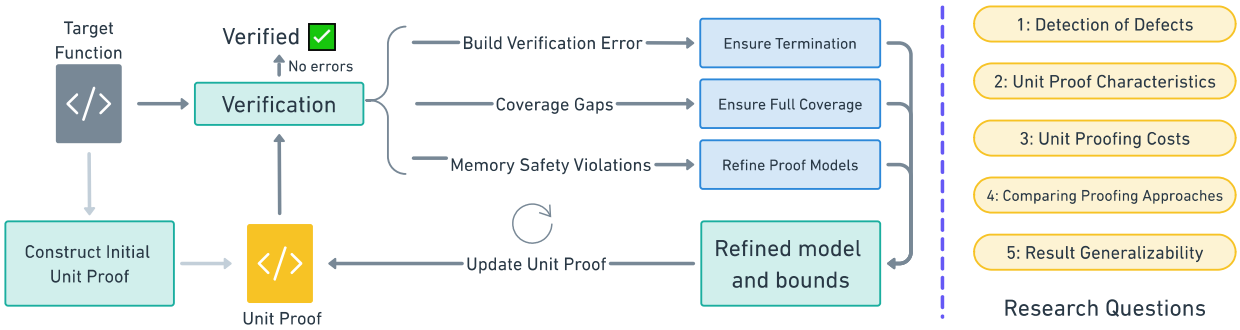

Figure 4.2. Study Methodology. We first develop unit proofs in 5 steps, iterating until full coverage and no memory safety violations. Then we evaluate the five research questions.

Software Selection

We conducted our study on four open-source embedded operating systems: FreeRTOS [60], Zephyr [42], Contiki-ng [74] and RIOT [155]. Embedded software is ideal for memory safety verification due to their use in safety-critical systems [11], lack of hardware mitigations [18], fuzzing challenges [54, 55] and high cost of fixing defects after deployment. The selected OSes are actively maintained and include diverse components (kernels, drivers, protocols, and libraries) that process external data, making memory safety defects in them

exploitable. We selected them among other OSes listed on www.osrtos.com [156] based on popularity (GitHub stars) and publicly disclosed vulnerabilities.

Functional Unit Selection

We selected functional units in the selected software with previously known defects. First, we identified defects in the National Vulnerability Database (NVD) that affect the selected software, has a memory safety CWE (*e.g.,* CWE-125, CWE-476, CWE-787)[3], and contained sufficient details for reproduction. This yielded 152 defects (Zephyr - 69, Contiki-ng - 47, RIOT-OS - 28 and FreeRTOS - 8). To reduce bias due to the varying number of defects, we selected 30 Zephyr defects and 25 Contiki-ng defects, as well as all RIOT and FreeRTOS defects, yielding a total of 89 defects.

To locate the affected functional unit for each defect, we reviewed the associated security advisories and patches to identify the lines of code where the defect manifests (an invalid memory is accessed) and was fixed. We identify the affected functional unit's entry as the closest function that reaches both locations. We selected 73 unique functional units, found across 22 distinct embedded software components (*e.g.,* network subsystem, file system, USB driver, etc) and 11 protocol implementations (*e.g.,* TCP, IPv6, DHCP, etc).

Bounded Model Checker Selection

We conducted all bounded model checking using the Ansi-C Bounded Model Checker (CBMC v6.3.1) [141]. CBMC is actively maintained and used in industry [122].

Experimental Setup

We used a Dell Precision 5820 Tower server: Intel(R) Xeon(R) W-2295 CPU @ 3.00GHz, 188GB of RAM.

[3]↑The NVD tracks security defects with a description, CVE identifier, and CWE category.

4.5.2 Systematically Developing Unit Proofs

We describe our approach, completeness criteria, and unit proofing steps. Figure 4.3 illustrates the steps.

Unit Proofing Approach

The key difficulty in unit proofing is modeling the unknown components (variables, functions) a unit depends on (§4.3.3). We use a *bottom-up strategy* called "angelic modeling" [157] and "assume-guarantee reasoning" [158], which derives models that ensure the target unit behaves correctly, verifies the unit using these models, and then validates the model assumptions. In this work, we systematically use the coverage report and error traces to model unknown components. A similar approach has been applied in prior works [157, 159, 160].

Alternatively, a *top-down strategy* first constructs accurate models of unknown components for verification, as seen in prior unit proofing reports [122, 128, 129]. However, it depends on existing specifications or project-wide knowledge, increasing its potential cost.

Completeness Criteria for Unit Proofs

We used the following objective criteria to assess the completeness of unit proofs. As prior works do not provide any metrics for assessing unit proofs, we derived these metrics from our experience.

1. *Graceful Termination:* The unit proof builds, executes and terminates after verification completes.

2. *Coverage:* The unit proof covers all reachable lines of code in the functional unit. Line coverage is commonly used for assessing unit tests [161, 162] and fuzzing campaigns [163, 164].

3. *Memory safety violations:* Our unit proofing philosophy relies on using memory safety violations in the error report to derive models of unknowns in the unit proof. Hence, we

consider the unit proof complete when all reported memory safety violations are either traced to unknowns and resolved using refined models or they are identified as real bugs and fixed.

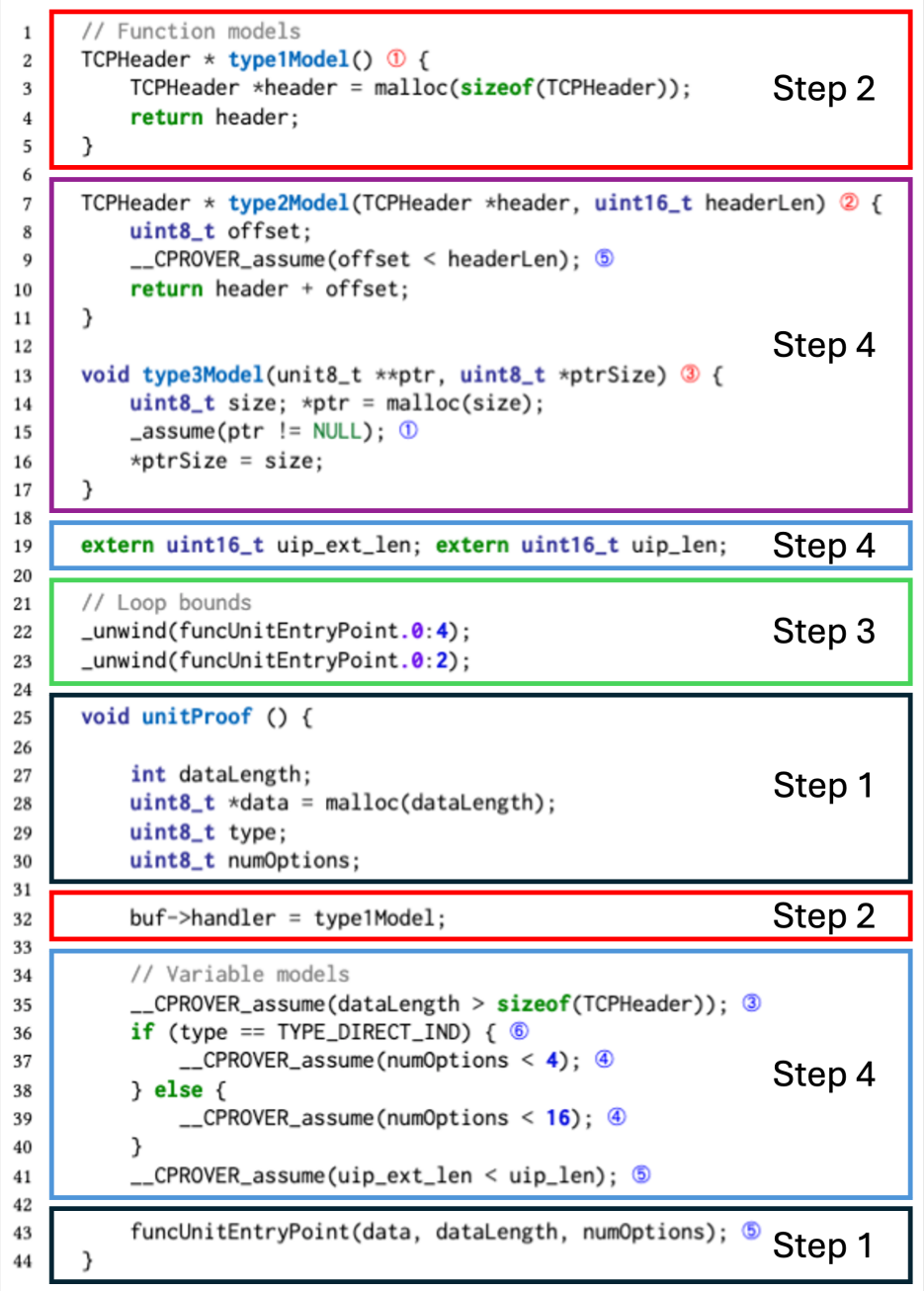

Figure 4.3. A systematically-developed unit proof. For realism, each element is taken from a real proof. In step 1, the initial unit proof contained only the black box. Initial verification timed out due to an unconstrained function pointer. In step 2, we resolved this (adding red box). The coverage report showed gaps due to insufficient loops. In step 3, we increased the bounds of affected loops (adding green box). Finally, the error report indicated errors caused by unknown variables and functions. In step 4, we resolved these via variable preconditions (blue box) and function models (purple box).

Step 1: Constructing an Initial Unit Proof

First, we set up the proof directory containing the proof harness and proof makefile. The initial proof harness is based on the CBMC's proof writing guide [165] and defines input variables, allocates necessary pointers, and invokes the target function (entry point of the functional unit). The makefile specifies build (*e.g.,* program configurations) and verification (*e.g.,* loop bounds) options. We reuse the Makefile from the FreeRTOS repository [166] and set all loop bounds to 1.

Step 2: Ensuring Termination

We build and execute the unit proof using the `make` command. However, the process may fail or lead to prolonged verification. Build failures typically result from missing variable definitions, header files, or incorrect configurations. We analyze error messages and resolve them accordingly. For prolonged verification or memory exhaustion, we identify the causes and apply appropriate interventions. Common interventions involved initializing function pointers, breaking recursion chains and replacing a complex called function with a simpler model.

Step 3: Ensuring Full Coverage

After verification, we identify uncovered code blocks from the coverage report. These coverage gaps were typically caused by insufficient loop bounds or data allocation, and fixed by increased loop unwinding on affected loops or setting minimum bound on data allocation.

Step 4: Refining Variable and Function Models

Memory safety violations, reported during verification, may be caused by unknown components and not represent actual defects. In such cases and following our angelic modeling approach (§4.5.2), we use the reported error traces to identify which unknown variable or function caused the error and derived or refined the model for that component.

For violations linked to input or global variables, we derive the minimal precondition necessary to resolve the issue. If violations stem from an undefined function's return value, we introduce a model for the function and constrain the return value. In instances where the unknown variable is initialized or validated in an undefined function (these functions had descriptive names like `prvAllowIPv4Packet` in FreeRTOS), we model the function's expected behavior or expand the unit's scope by including the file containing the missing function's definition. Examples of derived models are in Figure 4.3 and in the accompanying research artifact.

After deriving preconditions, we then analyze the functions producing these variables (*e.g.,* callers or undefined functions) to determine if the derived precondition can be violated. Without automated tools, we prioritize unlikely preconditions based on experience and intuition. If a violation was caused by an unvalidated local field (*e.g.,* field read from an input data) or the derived precondition can be violated, we report it as a defect to maintainers.

4.5.3 Answering Research Questions

RQ1: Detection of Memory Safety Defects

We evaluate whether unit proofs developed systematically, using the approach in §4.5.2, can detect memory safety defects. We re-introduce known memory safety defects in their corresponding functional units, assess the defect detection rate and investigate reasons for non-detection.

Reintroducing Known Defects: Similar to prior work [27, 167], we reintroduce the defects selected in §4.5.1 by identifying the patches that fixed them and reverting these changes in the affected embedded OS. Identifying patches involved reviewing security advisories and relevant GitHub pull requests to understand how each defect manifests and was fixed. Since most defects were addressed by adding validation, reversion typically involved removing or falsifying these checks. Initially, we attempted reverting to pre-fix versions but encountered frequent build failures due to unavailable, incompatible, or outdated dependencies, making this approach impractical and difficult to scale.

Assessing Defect Detection: To determine if a defect is exposed, we execute the corresponding unit proof and analyze the error report for related memory safety violations. If no violation is reported, we re-examine the affected function and execution logic, modifying the unit proof as needed to expose the defect. We report any required interventions and reasons for non-exposure.

De-biasing: Clearly if an analyst knows the details of a defect, their approach may be biased. To mitigate this, we used a blind approach when creating the unit proofs. One author identified the functions and introduced the defects while all unit proofs were developed by other authors who received only the affected functions. The remaining issue is that the proof-writers knew that the functions were indeed defective, which may have made them more persistent. We note two factors that mitigate this: (1) our systematic approach gave them clear stopping criteria, causing them to miss some defects, and (2) our approach caused them to find new defects in 19 of the 73 functions they verified, indicating that they kept an open mind.

RQ2: Characteristics of Unit Proofs

We characterize unit proofs developed systematically by their size and contents. A program's size and contents impacts its understandability [168, 169] and likelihood to contain errors [170]. Hence, smaller, simpler unit proofs will be easier for software engineers to develop, understand, and maintain. Understanding the models and loop bounds required in unit proofs can also inform future research on automating their derivation. We measure *size* by counting lines of code and number of models in the unit proof. To analyze *structure*, we categorize model types and loop bounds used in the unit proof.

RQ3: Time to Develop and Execute Unit Proofs

We evaluated the time required to develop and execute unit proofs using the proposed methodology. The time to develop unit proofs will affect adoption costs and influence adoption decisions, while execution time will impact the feasibility of integrating unit proofs

during continuous software development. Given our iterative approach, the execution time will also influence the development time and the overall verification experience.

For 21 unit proofs [4], we measure and report development time for each proof and each step of the process (§4.5.2) and identify the reasons for longer durations. We then measure verification time for all unit proofs by modifying the build file to report the duration of the `cbmc` execution command, and excluding compilation time. Finally, we examine how execution time correlates with program size and the complexity of the solved formula to assess their potential as predictors of verification time.

RQ4: Systematic vs. Expertise-Based Unit Proofs

We evaluate how unit proofs developed systematically (§4.5.2) and with objective completeness criteria (§4.5.2) compare to those developed by project experts. Among the selected software, only FreeRTOS contained existing unit proofs. We identified 8 FreeRTOS unit proofs covering the same functional units selected in §4.5.1. These proofs, developed in 2020, likely followed the process described by Chong *et al.* [122]. We then compared the characteristics (§4.5.3) and execution time (§4.5.3) of the FreeRTOS unit proofs with the corresponding systematic unit proofs. Finally, we assessed how the different development approaches influenced these results.

RQ5: Generalizability to Other Functions

RQ1 to RQ4 were evaluated on a subset of functions from the selected software, which may not represent the remaining functions, limiting generalizability. To assess this potential bias, we compare the evaluated subset to the remaining functions in each software. Given that verification effort correlates with program size and complexity [131, 132], we measure these metrics using the Lizard tool [171]. We collect lines of code and complexity data for all functions and created two datasets: one for functions with unit proofs and another for all

[4]↑These were the last unit proofs we developed. We used the first 52 proofs to develop the necessary unit proofing experience.

functions. We then compare their size and complexity metric distributions using statistical plots.

4.6 Results

In this section, we present our results by RQ.

4.6.1 RQ1: Detection of Memory Safety Defects

Table 4.1. Known and new memory safety vulnerabilities detected using unit proofs.

Defect Status	**Count (%)**
Total Exposed	74 (83%)
Exposed systematically	66
Exposed by increasing BMC bounds	8
Total Not Exposed	15 (17%)
Memory Exhaustion	10
Others	5
Total known defects	89 (100%)
New Defects Found	19
Out-of-bound write	5
Out-of-bound read	12
Null pointer dereferencing	1
Arithmetic underflow	1

Table 4.2. Table showing the proportion of known vulnerabilities that were exposed by unit proofing, and the number of new vulnerabilities found.

Defect Status	**Count (%)**
Total Exposed Systematically	66 (74%)
Total Exposed with Interventions	8 (9%)
Increasing loop bounds	4
Expanding unit's scope	3
Increasing bounds on string length	1
Total Not Exposed	15 (17%)
Memory Exhaustion (before adding defect)	4
Memory Exhaustion (after adding defect)	6
Substantially modified code	1
Undefined API	2
Requires timer event	1
Unknown	1
Total known defects	89 (100%)
New Defects Found	19
Out-of-bound write	5
Out-of-bound read	12
Null pointer dereferencing	1
Arithmetic underflow	1

Table 4.2 presents the results of using systematically developed unit proofs to detect memory safety defects. Of the 89 defects analyzed, 66 (74%) were exposed using systematic unit proofs, 8 (9%) required increasing BMC bounds, and 15 (17%) remained undetected. Also, 19 new defects were exposed using the systematic unit proofs.

Investigating Defects Requiring Additional Interventions: Among the five defects requiring increased bounds (*e.g.,* CVE-2024-32017), four involved copying strings into fixed-size

buffers. Our approach (§4.5.2) adjusted BMC bounds only when it was necessary to increase coverage, and since the string-copying operations were already covered, further increases were not triggered. One defect (CVE-2023-6749) also required a string length bound of 32, exceeding our default of 20. The last defect (CVE-2020-14935) also involved accessing a fixed size buffer in a loop and was not also exposed because the function's coverage only required a lower loop bound.

Three other defects (*e.g.,* CVE-2024-4785) were missed due to our systematic approach limiting unit scope to functions within a single source file. Since the invalid memory access occurred in children functions in separate files, detecting them required expanding the unit's boundaries to include the relevant external function.

Investigating Defects Not Exposed: Ten defects were undetected because the bounded model checker (BMC) ran out of memory before finding a solution. Four occurred during initial unit proof development, before the defect was reintroduced. In three cases, the issue involved memmove, which, when removed, resolved the problem. The fourth case involved an undefined function, whose removal also resolved the problem, though the exact reason remains unclear. In the remaining six cases, detecting each defect required expanding the unit's boundary to include an additional file. However, this expansion led to memory exhaustion, preventing verification.

Two defects stemmed from unsafe sprintf usage, but since CBMC does not support sprintf, they were not exposed. Two others were missed due to significant changes in the target function, as the defect manifests during timer event processing, which CBMC does not support. The cause of the final undetected defect remains unknown.

Exposing New Defects: Using systematically developed unit proofs, we uncovered 19 new defects: 9 in RIOT, 5 in Zephyr, and 5 in Contiki-ng. No defects were found in FreeRTOS, likely because the functions we verified had already been verified by the FreeRTOS team. All defects were reported to maintainers following their responsible disclosure policies. Five have been fixed and assigned CVE identifiers (with CVSS scores above 8.0 indicating high severity). The rest are under investigation. Details of these defects are included in our artifact.

Definition 4.6.1. RQ1 Finding: Unit proofs developed systematically detected 74% of recreated defects and 19 new defects. Increasing BMC bounds exposed additional 9%. 11% of defects not exposed due to memory exhaustion during verification.

4.6.2 RQ2: Unit Proof Characteristics

RQ2 characterizes the size and components of systematically developed unit proofs.

Characterizing Unit Proof Sizes

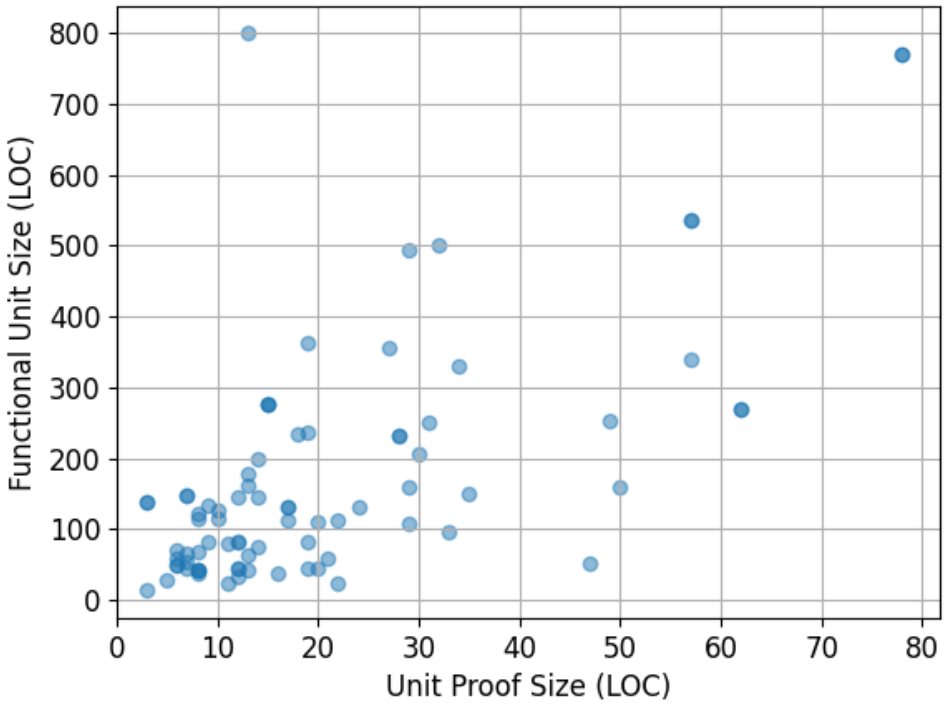

Figure 4.4. Plot showing unit proof sizes vary by unit sizes.

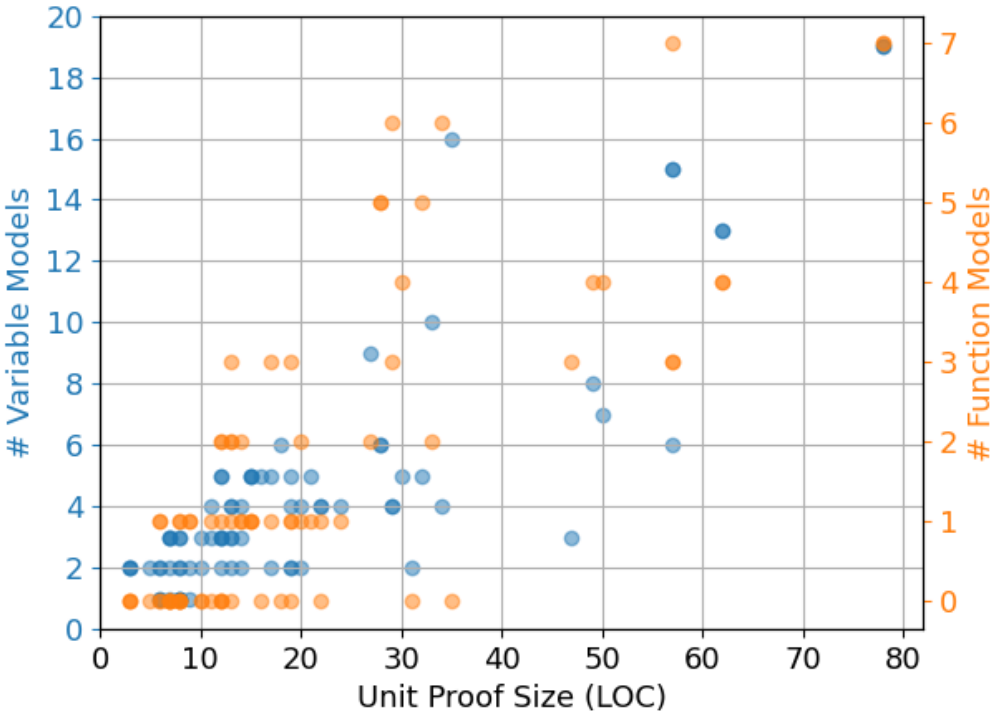

Figure 4.5. Plot showing the number of variables and models in unit proofs of different sizes.

Figure 4.4 illustrates the size of unit proofs across different functional units. Notably, proofs had mean size of 20 lines of code, with only five exceeding 50 lines. The data indicates a direct correlation between unit proof size and functional unit size: larger units require proportionally larger proofs, averaging approximately one-tenth of the unit's size.

Figure 4.5 shows the number of variable and function models in unit proofs of varying sizes. On average, unit proofs contain 5 variable and 2 function models, with model count increasing with proof size. These findings indicate that realistic functional units in embedded software typically require small unit proofs, typically less than 50 lines, and only a few models.

Characterizing Variable and Function Models

Figure 4.3 presents a realistic unit proof, highlighting the types of variable and function models it contains. Table 4.3 shows that pointer-not-null ①and allocation size ③models account for 76% of variable models in unit proofs. As discussed in §4.7.2, these we observed that these models can be derived statically. In other cases, recognizing these model types helped the authors determine how to model unknown variables to resolve reported violations.

Table 4.4 categorizes function models into three types: Type 1 and Type 2 model only the function's return value, while Type 3 models include input behavior. Among 113 modeled functions, 76% required only return value modeling, typically so it returns valid pointers of constrained sizes. This demonstrates that memory safety verification primarily relies on return value modeling, reducing the cost compared to verification tasks requiring full semantic behavior modeling.

Table 4.3. Prevalence of different variable models across unit proofs. The different model types are numbered and illustrated in Figure 4.3.

OS	**# Unit Proofs**	**# models**	**Pointer not null ①**	**Pointer and offset ②**	**Allocation size ③**	**Integer range ④**	**Integer r/ship ⑤**	**Conditional modeling ⑥**
Zephyr	25	159	87	8	31	15	16	2
Contiki-ng	18	51	19	2	9	11	8	2
RIOT	22	132	78	3	32	9	8	1
FreeRTOS	8	25	17	0	6	2	0	0
Total	73	367	201 (55%)	13 (4%)	106 (21%)	37 (10%)	32 (9%)	5 (1%)

Table 4.4. Characterizing function models in unit proofs. Type 1 models model only the function's return value based on the type. Type 2 models model the return type using program-specific semantics. Type 3 models input arguments together with return values. These models are numbered and illustrated in Figure 4.3.

OS	# models	Type 1 ①	Type 2 ②	Type 3 ③
Zephyr	37	29	5	3
Contiki-ng	21	14	6	1
RIOT	40	34	4	2
FreeRTOS	15	9	4	2
Total	113	86 (76%)	19 (17%)	8 (7%)

Characterizing Loop Bounds

Table 4.5. Characterizing loops based on exit condition.

Loop condition type	Count	b = 2	b = 3	max b
Constant	33	9	12	65
Data length	13	10	3	3
Linked-list length	18	16	2	3
strlen/strcpy/memcmp	21	0	0	65
Others	7	5	1	5
Total	92	40 (43%)	18 (20%)	65

Table 4.5 categorizes loops requiring custom bounds based on their exit conditions. 58 (63%) loops required unrolling only 2 or 3 times. The required bound varied by loop

type: loops with a static number of iterations required full unrolling, while loops processing variable-length data or linked lists typically needed a bound of 2 or 3. String-handling loops required bounds matching the maximum string length, and loops in `memcmp` implementations required bounds corresponding to the `memcmp` size argument.

Definition 4.6.2. RQ2 Finding: Systematic unit proofs are small, averaging 20 lines of code, 5 modeled variables, and 2 modeled functions. 76% of variable models can be derived statically, and 77% of function models return unconstrained values. 63% of loops require bounds <= 3, with the bound depending on loop type.

4.6.3 RQ3: Unit Proof Development and Execution Time

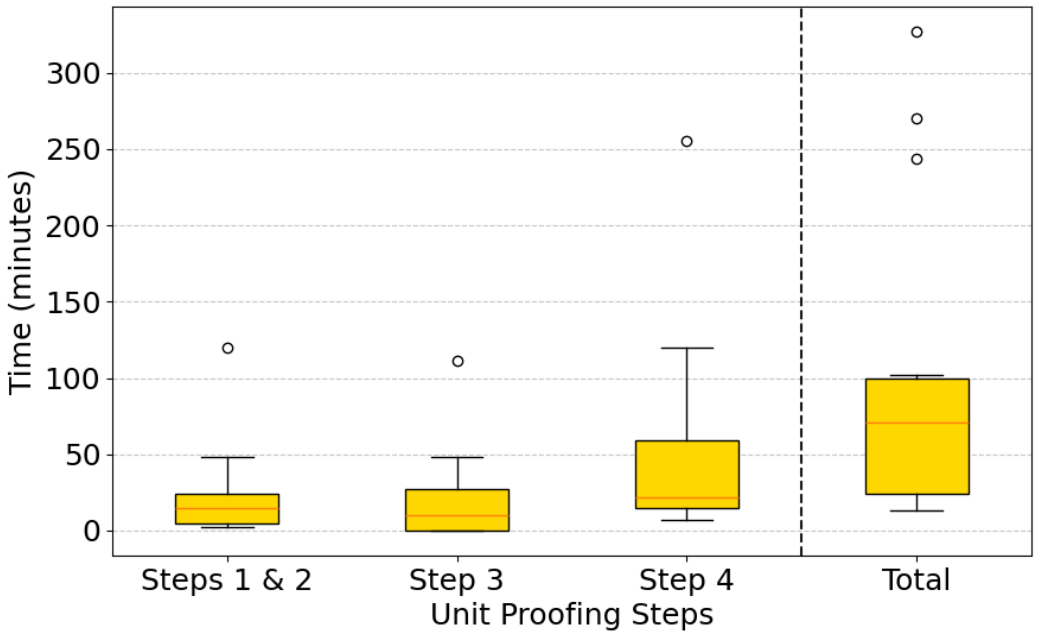

Figure 4.6. Time it takes to develop unit proofs.

Unit Proof Development Time

Figure 4.6 illustrates the time spent on unit proof development, covering setup (steps 1 & 2), coverage improvement (step 3), and model derivation (step 4). The median unit proof

took 72 minutes, with significant time devoted to modeling unknown variables and functions based on reported violations.

Delays in steps 1 & 2 were mainly due to missing headers or undefined symbols when compiling units in isolation. Step 3 took longer for units with multiple loops, as our iterative approach derived loop bounds one at a time to improve coverage. Step 4 delays mostly arose from units with substantially larger sizes and which which required more modeled variables. For instance, the two unit proofs with the longest Step 4 duration (4 hours 15 mins and 2 hours) verified 412 and 766 lines of code respectively and required 17 and 19 variable models respectively.

The reported durations exclude the one-time setup overhead for each embedded OS, which involved adapting to different build instructions, header paths, and configuration definitions. For example, Zephyr's custom build system (West [172]) relied on autogenerated headers and custom compile flags, causing errors when compiling files in isolation. We resolved this by modifying our Makefile to build a full Zephyr kernel, generating headers and extracting compile flags from compile_commands.json. Once resolved, this approach was reused for other unit proofs within the same OS.

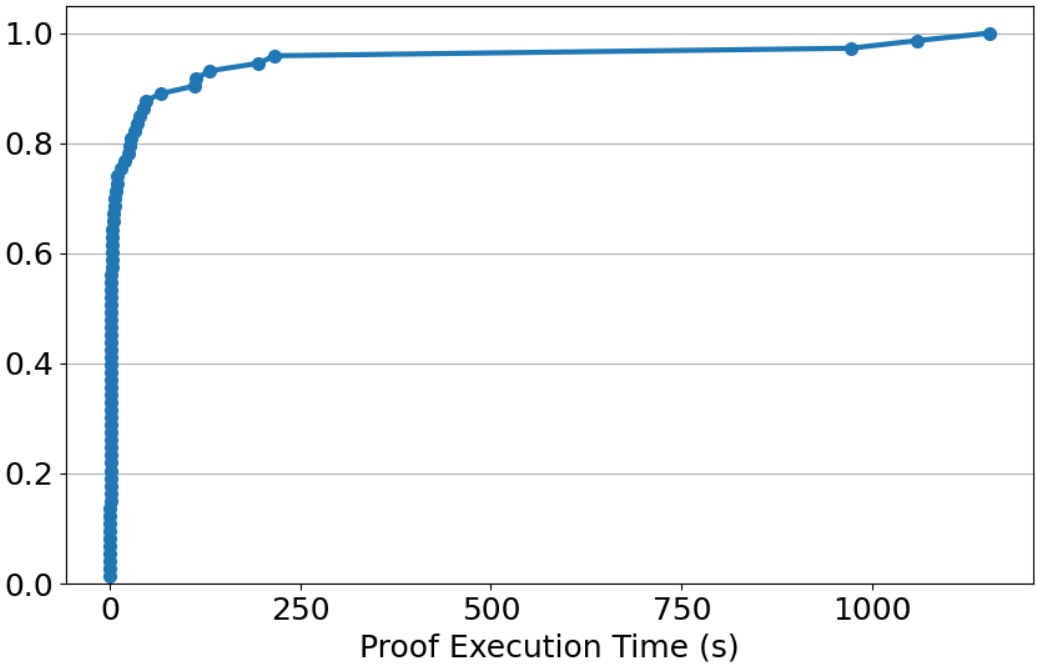

Figure 4.7. Cumulative density function (CDF) showing the time it takes to execute unit proofs.

Unit Proof Execution Time

Figure 4.7 shows unit proof execution times, with 87.67 % completing in under 60 seconds and only 5 taking longer than 3 minutes. We attribute these fast times to minimal loop bounds, limiting the scope of units, and handling complex semantics like function pointers and recursion (§4.5.2). This supports the feasibility of integrating unit proof execution into development or code commits for continuous verification [173]. When compared, we also found that execution time correlated more strongly with verification formula size (R^2 = 0.852) than program size (R^2 = 0.029) (Data and charts in artifact). Hence, they can be used to predict the verification time and help when determining appropriate size of a functional unit.

Definition 4.6.3. RQ3 Finding: On average, unit proofs take 82 minutes to develop and 61 seconds to execute. The median proof required 72 minutes to develop, 25 seconds to

execute, and verified 185 lines of code. Verification formula size accounts for 85% of the execution time.

4.6.4 RQ4: Systematic vs. Expertise-Based Proofs

Figure 4.8. Comparing systematically developed FreeRTOS proofs with those developed by FreeRTOS engineers.

Figure 4.8 compares the unit proofs developed systematically with the existing unit proofs in FreeRTOS. We find that the systematically developed proofs had fewer files in scope, fewer modeled variables, lower loop bounds, faster execution, and better coverage. Systematic proofs included additional files only when necessary to resolve violations,. However, this preventing three defects from being detected initially (Table 4.2). In contrast, FreeRTOS proofs included all reachable files in the unit proof's scope, increasing the functional unit size and enhancing verification fidelity. However, this also increased the number of required variables models, proof execution time and potentially, modeling effort.

Figure 4.8 also shows that existing FreeRTOS unit proofs achieved less coverage compared to the systematic proofs. In one case (ProcessReceivedTCPPacket_harness.c), the FreeRTOS proof achieved only 10.76 % coverage due to an error in the unit proof model, where an incorrect assumption fixed the packet length to an invalid value. This error could have been detected if coverage had been used to assess proof completeness. A similar issue was found in DNS_ParseDNSReply_harness.c, which had 5.75 % coverage, indicating a broader concern. Additional inconsistencies were also observed. For instance, while some

proofs (*e.g.,* `ProcessICMPPacket _harness.c`) modeled packet buffers with variable lengths and an upper bound, others (*e.g.,* `DNS_ParseDNSReply_harness.c`) used fixed sizes. Prior work [130] has shown that using fixed sizes can prevent defects triggered by smaller packets from being detected.

Definition 4.6.4. RQ4 Finding: The systematically developed unit proofs modeled fewer variables, executed faster and achieved better coverage. Observed errors and inconsistencies in existing FreeRTOS proofs indicate the need for a uniform and systematic approach.

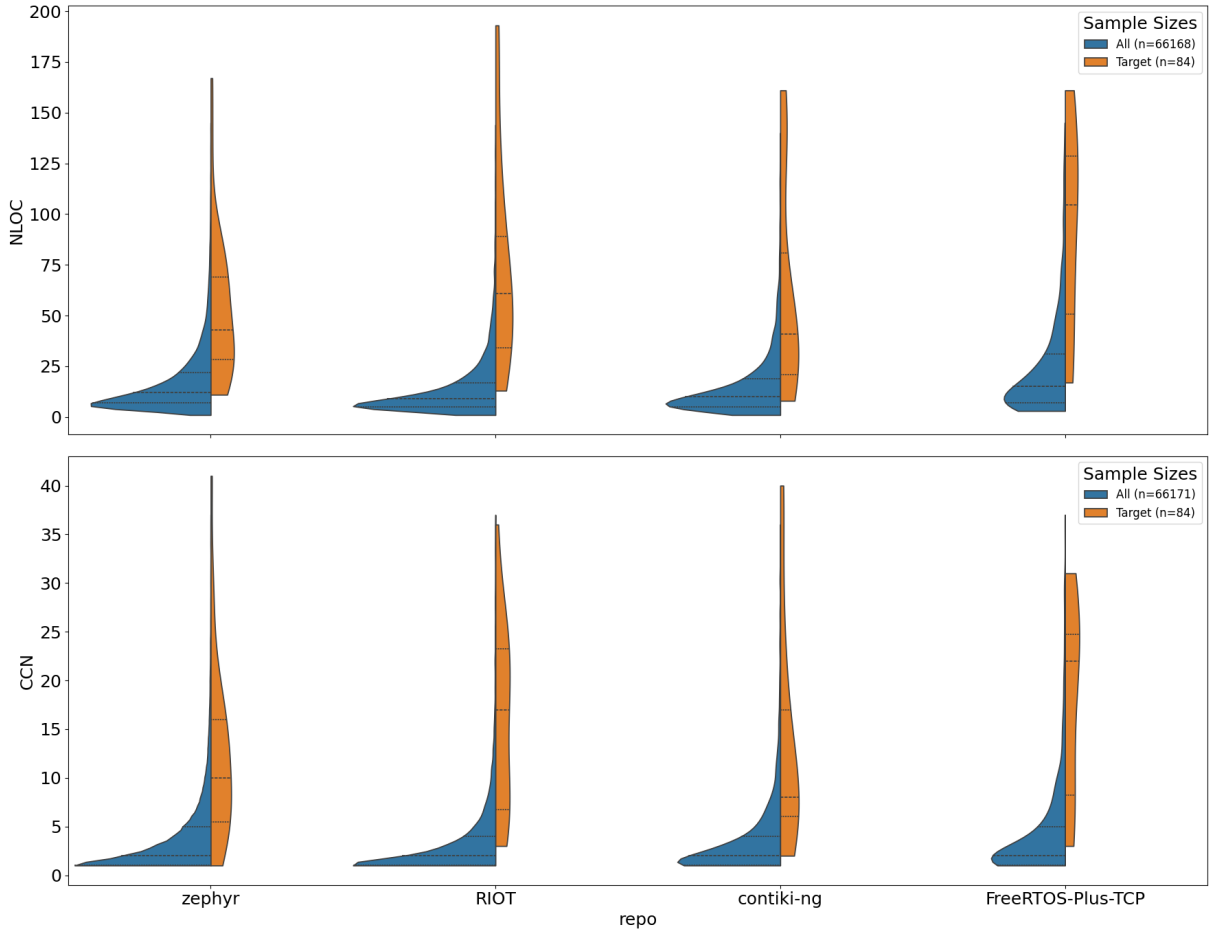

Figure 4.9. Comparing the size (top) and complexity (bottom) of verified functions with the functions in each embedded OS. CCN: Cyclomatic complexity number. NLOC: Number of lines of code.

4.6.5 RQ5: Generalizability of Results

Figure 4.9 shows that the functions we developed unit proofs for are among the larger (top) and more complex (bottom) ones in each embedded OS. Given that unit proof size (Figure 4.4), development time, and execution time (§4.6.3) scale with function size, we infer that the remaining functions in each embedded OS will require similarly sized or smaller unit proofs, fewer models, and overall similar or less effort than those reported in Figure 4.6.

Definition 4.6.5. RQ5 Finding: The developed unit proofs cover the larger and more complex functions in each embedded software, demonstrating the generalizability of our approach.

4.7 Discussion

In this section, we discuss the broader implications of our results.

4.7.1 Our Cost-Benefit Analysis for Unit Proofing

A key question asked by engineering leaders is: *Should we adopt technique X?* This paper provides empirical data on the costs and benefits of unit proofing to support informed decisions. Table 4.2 shows that following the systematic approach in §4.5.2, unit proofing detects up to 74% of memory safety defects. With minor adjustments — such as unrolling string-handling loops based on usage and incrementally expanding unit scope — this increases to 83%. Additionally, RQ1 results (§4.6.1) demonstrate that unit proofing supports incremental verification, enabling teams to uncover defects without verifying the entire code base. By writing unit proofs for 73 functions, we identified 19 new memory safety defects (Table 4.2). Each unit proof contributes value, either by exposing a defect or providing safety guarantees for the verified unit.

On cost, Figure 4.6 shows that the median unit proof took 72 minutes to develop and 25 seconds to execute, verifying 185 lines of C code (the `_parse_options()` function in RIOT [174]). At this rate, an engineer could verify approximately 1200 lines in one workday (8 hours). This is significantly faster than other verification efforts, often measured in

person-years [58]. This reduced time cost, driven by our systematic approach, requires minimal project-wide knowledge, allowing developers to write and verify unit proofs in isolation. For comparison, the AWS team, using their approach to unit proofing, reported that verifying 1500 lines of code took an engineer a month (or 75 lines of code per workday). Considering the high cost of memory safety defects [7] — which lead to outages [5], security vulnerabilities [1], and attacks [3] — we conclude that unit proofing offers a cost-effective strategy for mitigating these defects during development. Because our unit proofing approach can be applied incrementally, it need not be applied to an entire codebase, and engineers may focus on components that are particularly vulnerable (*e.g.,* user-facing) or unprotected by other means (*e.g.,* lacking hardware protections).

4.7.2 Future Work: Tooling for Unit Proofing

Our approach was conducted entirely manually. Tooling would further reduce costs and accelerate adoption. Our empirical measurements can guide tool designs. We give three examples.

(1) Modeling Unknowns: The most time-consuming aspect of unit proofing is deriving models for unknown variables and functions (Figure 4.6), many of which could be inferred statically during the initial proof creation. For instance, 55% of derived variable models were preconditions assuming a pointer was non-null (Table 4.3), which could be identified through static analysis of pointer usage [47, 175, 176]. Similarly, 22% of derived models constrained buffer sizes and could be inferred by analyzing indexing operations. While static techniques can be imprecise, tools can be tuned to generate models only when confidence is 100%, ensuring accuracy and improving efficiency.

(2) Loop Bounds: Tooling can also assist in deriving minimal loop bounds. Fixed-iteration loops require bounds based on iteration counts (§4.6.2), while string-handling loops should be bounded based on usage (§4.6.1). Together, these categories account for 59% of loops requiring custom bounds (Table 4.5) and could be inferred through static analysis of loop exit conditions and buffer accesses.

(3) Analyzing Error Traces: Additionally, automated tooling can analyze error traces to determine whether failures stem from unknown variables or functions and generate the necessary models to resolve these issues. There are algorithms for deriving preconditions for functions [159, 177–179], but they are designed for programs already in formal logic or abstract states rather than in regular programming languages. Adapting these algorithms to support model generation for unit proofs presents another promising yet unresolved research direction.

Integrating these tooling advances into development environments (IDEs) would enable engineers to create unit proofs alongside software development, similar to writing unit tests. With efficient tooling and fast execution times, engineers could verify functions in isolation before merging new code, ensuring higher reliability. We also leave to future work, the feasibility of applying generative AI technologies such as LLMs to unit proofing. We piloted these tools but find them currently weak, perhaps due to their probabilistic nature and the minimal real-world examples in their training data.

4.8 Threats to Validity

Construct Validity: Multiple operationalizations of compositional bounded model checking are possible. We specifically studied one unit proofing strategy. Our findings may not generalize to other approaches (§4.3.3).

Internal Validity: §4.5.3 discusses how we mitigated bias when creating unit proofs. To reduce bias when characterizing unit proofs, two authors collaboratively developed the taxonomy, independently characterized the unit proofs, and then harmonized their results. To reduce bias in reporting characteristics and cost (RQ2–3), a different author reviewed and ran all unit proofs, automatically extracting the reported quantitative data (§4.6.3).

External Validity: Our study evaluated unit proofing on functions from embedded operating systems. This raises concerns about generalizability. *On what kinds of functions do our results hold?* RQ5 demonstrates that our study included larger, more complex functions, indicating our results represent an upper bound on unit proof size and cost in embedded software. *Would our results hold on other embedded software?* Our approach was effective

across the 22 diverse components in the evaluated embedded OSes, and the uniformity of functions in Figure 4.9 suggests broader generalizability to other embedded software. *Would our results hold if other engineers applied unit proofing?* Part of the value of our work is that we followed a systematic approach with clear stopping criteria (see §4.5.2 and our artifact). Our approach was clear enough that four authors could contribute to proof development, reducing individual bias. We believe that other engineers following our approach would achieve similar results. *What about other (non-embedded) software?* The main risk is that the constraints of embedded software lead to similar software architectures and patterns, *e.g.,* event-driven approaches and reliance on global variables. We cannot comment on generalizability to software designed with other constraints.

5. SAFETY-ORIENTED UNIT PROOF GENERATION FOR MEMORY-SAFETY ASSURANCE

5.1 Summary

Chapter 4 showed that high-assurance component-level bounded model checking can become more practical when unit proofs are constructed using a systematic process. However, that process remains manual: engineers must choose the verification scope, loop bounds, and environment model that make verification both tractable and meaningful. These choices determine whether bounded model checking exposes genuine memory-safety errors or provides useful evidence that such errors are absent under explicit assumptions.

This chapter introduces safety-oriented unit proofs, reusable component-level verification artifacts that prioritize the program behavior needed to check memory-safety properties rather than faithfully modeling all execution behavior. It addresses the remaining automation barrier by deriving verification scopes, loop bounds, and environment models from the target memory-safety properties and by validating each generated choice before using it in verification.

Methodology: This chapter presents AutoSOUP, a system that automatically constructs safety-oriented unit proofs for C software. AutoSOUP combines deterministic program-analysis workflows with LLM-driven subtasks in an *LLM-as-function-call* architecture. Its deterministic orchestration defines the task objectives, invokes LLMs for bounded code-understanding and synthesis tasks, and checks the resulting scope, bounds, and assumptions before incorporating them into the unit proof. AutoSOUP introduces three techniques for deriving verification choices: resource-aware scope widening, property-guided loop-bound selection, and context-aware environment refinement. We evaluate AutoSOUP on 177 components from four embedded real-time operating systems, using generated-proof utility, vulnerability exposure, technique ablations, cost, and comparison with expert-written unit proofs.

Findings: AutoSOUP successfully produces valid unit proofs and verifies 93% of candidate targets. On recreated vulnerability benchmarks, it exposes 66.7% of evaluated CVEs, outperforming both a general-purpose coding-agent baseline and a recent memory-safety

verification baseline. It also identifies 20 previously unknown externally triggerable vulnerabilities, including out-of-bounds writes with denial-of-service or arbitrary-code-execution potential. The ablation results show that AutoSOUP's three techniques are complementary, and the comparison with expert-written proofs shows that AutoSOUP uses simpler, auditable environment assumptions while achieving comparable verification outcomes. Together, these results show that safety-oriented unit proofs can automate a practical component-level path to memory-safety assurance for realistic embedded software.

Statement of Attribution: The work in this chapter appears in a technical report [28]. I am the lead author and claim primary intellectual ownership of the work and the content of this chapter.

5.2 Introduction

Memory-safety errors remain a persistent source of zero-day exploits in low-level software [180, 181]. These errors enable denial-of-service, data-theft, and remote-code-execution attacks [23]. Their risk is especially acute in embedded systems, which often lack standard hardware protections [18] and are difficult to analyze effectively with techniques such as fuzzing [54, 182]. After decades of recurring exploits and failures [23, 34, 181, 183], government agencies and industry organizations increasingly advocate or mandate methods that provide stronger memory-safety guarantees [8, 9, 127, 184].

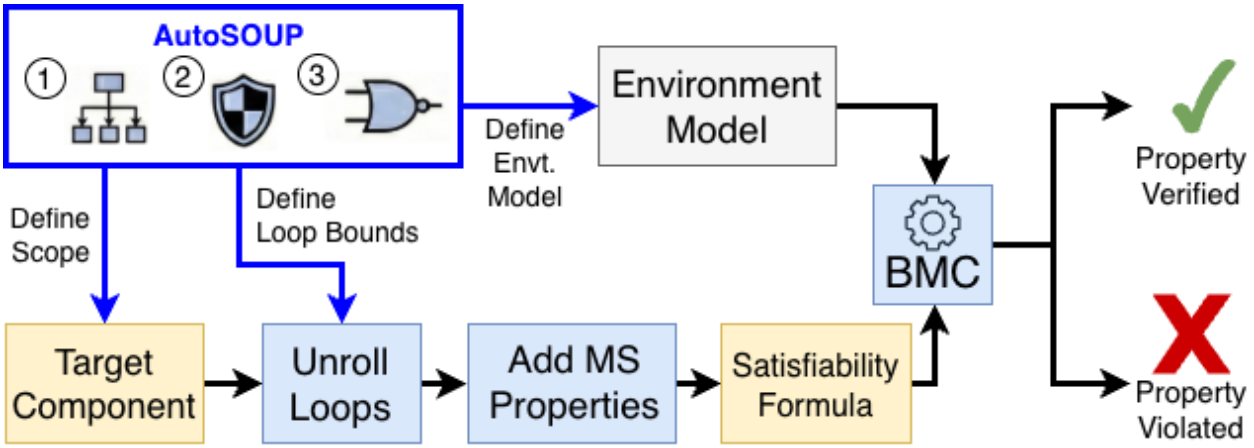

Figure 5.1. AutoSOUP automatically identifies the functions to include in the verification scope, loop bounds, and environment model for which the resulting component-level memory-safety verification completes and provides useful guarantees of memory safety.

Formal memory-safety verification can provide such guarantees, but existing techniques remain difficult to apply at scale. Deductive verification [185] can prove memory safety [25, 186], but it often requires specialized verification expertise, domain knowledge, and manual proof guidance [56, 58]. Bounded model checking (BMC) [139, 187] offers a more automated alternative, but whole-program BMC does not scale to large software systems. Recent industry case studies [122, 128, 129] favor a decomposition approach, wherein engineers verify components in isolation using *"unit proofs"* [188] that specify a component's verification scope, loop bounds, and environment models. However, constructing these unit proofs remains challenging and error-prone: engineers must choose scopes, bounds, and models that are strong enough to expose genuine memory-safety errors, but constrained enough to keep verification tractable. AWS's report [122] suggests a substantial engineering cost, with a verification rate of 1500 lines of code in one person-month. No existing work automates these choices in a way that supports practical component-level memory-safety verification for real C software.

To address this gap, this paper introduces AutoSOUP (Figure 5.1), a hybrid system for constructing unit proofs that enable useful memory-safety verification for embedded software. The key idea is the notion of *memory-safety-oriented unit proofs*: reusable verification artifacts that prioritize the scope, bounds, and environment models needed to preserve safety-

relevant memory behavior, rather than faithfully model all execution behavior. AutoSOUP derives these choices using three techniques: resource-aware scope widening, property-guided loop-bound selection, and context-aware environment refinement. To make this automation reliable and generalizable, AutoSOUP uses a hybrid *LLM-as-function-call* architecture [189, 190]. Rather than asking an LLM to synthesize an entire proof artifact, AutoSOUP uses deterministic program-analysis workflows to drive the construction process and invokes LLMs for controlled tasks with explicit validation criteria. As a result, each verification choice incorporated into the unit proof is tied to a specific algorithmic objective and validated before use.

We evaluate AutoSOUP on four embedded real-time operating systems. AutoSOUP successfully produces valid unit proofs and verifies 93% of candidate targets, exposing 66.7% of evaluated vulnerabilities—38.7% and 28.5% more than the next best-performing baselines. Each of our techniques contributes to the final verification results by improving the derived bounds and environment models. Moreover, AutoSOUP's unit proofs use simpler and more general environment assumptions than fidelity-oriented expert-written counterparts while achieving comparable verification outcomes.

In summary, this paper makes the following contribution:

- We propose *safety-oriented unit proofs*, a formulation of component-level memory-safety verification artifacts; and techniques for scope, bounds, and environment models.
- We operationalize the *LLM-as-function-call* architecture for automating auditable and justifiable unit-proof construction.
- We design and implement AutoSOUP to automatically construct safety-oriented unit proofs for C-language software.
- We evaluate AutoSOUP on 177 components from four widely used embedded operating systems and report its utility, effectiveness, and cost for memory-safety verification.

Significance: AutoSOUP is the first system to automate memory-safety verification through the construction of unit proofs that enable component-level bounded model checking. Our results show that AutoSOUP applies to substantial real-world codebases, provides useful memory-safety assurance, and exposes real security vulnerabilities. Automated

memory-safety verification can be practical enough to integrate into software-development workflows and help prevent memory-safety vulnerabilities before deployment.

5.3 Background

5.3.1 Memory-Safety and Verification

Memory safety means that all memory accesses performed by a program are valid under the semantics of the language [23]. Accesses must refer to memory that has been properly allocated, remain within the bounds of the target object, and occur during that object's lifetime [31, 191, 192]. A memory safety error violates these conditions [31, 193], either spatially or temporally [23]. Violations of these conditions cause memory corruption and remain a major source of software failures, security vulnerabilities, and exploits [3, 6, 7, 181].

Memory-safety verification uses formal methods to establish that a program cannot perform invalid memory accesses, subject to explicit modeling assumptions. Compared with testing and bug-finding techniques, its goal is not only to expose errors, but also to provide assurance that specified classes of memory violations are absent within the verified program.

Two approaches have been especially important for memory-safety verification. *Deductive verification* proves memory safety by reasoning over program behavior using specifications of program state, memory, and invariants [25, 160]. It can provide strong guarantees over all executions, but it requires substantial expertise and manual proof effort. In contrast, *Bounded Model Checking (BMC)* [194] checks memory-safety properties automatically by exploring executions within explicit bounds.

```
size_t get_record_count() {
  // Complex logic returning [0,10]
  return count;
}

int handle_record(size_t i) {
  ...
}

void process_record(uint8_t *dst) {
  size_t n = get_record_count();
  // Bug!! should be i < n
  for (size_t i = 0; i <= n; i++) {
    dst[i] = handle_record(i);
    assert(isValidObject(dst));
    assert(ObjectSize(dst) > i);
  }
}

void caller() {
  int dst[10];
  process_record(dst);
}
```

```
Scope = {process_record, handle_record}.
Loop bound: {process_record.0: 11}

size_t get_record_count_m1() {
  uint8_t ret = nondet_int();
  assume(ret < 10);
  return ret;
}

size_t get_record_count_m2() {
  uint8_t ret = nondet_int();
  assume(ret <= 10);
  return ret;
}

void harness(void) {
  uint8_t dst_size = nondet_int();
  uint8_t *dst = malloc(dst_size);
  assume(dst != NULL);
  process_record(dst);
}
```

Listing 5.1: *Left*: Program instrumented with memory-safety properties and its calling context. *Right:* Unit proof with verification choices for the `process_record` component.

5.3.2 Bounded Model Checking and Unit Proofing

Tools for Bounded Model Checking (BMC) [141, 142] specify program properties as Boolean assertions over variable states, automatically check these assertions using constraint-solving tools, and report counterexample traces [140]. For memory-safety verification, BMC instruments the program with memory-safety assertions, unrolls loops and recursion, translates the resulting bounded program into a satisfiability formula, and applies constraint solvers. If no violating trace exists, BMC proves that the instrumented properties hold. If a violation exists, BMC returns a counterexample.

Listing 5.1 illustrates how BMC can be used to verify a component's memory safety. An engineer verifies a connected subset of a program to keep verification tractable (*verification scope*); for example, they may verify `process_record` and `handle_record` while excluding

the `get_record_count` callee. The engineer must also choose the maximum number of times to unroll loops (*loop bounds*), because BMC operates only on loop-free programs, and provide assumptions about the behavior of callers and callees outside the chosen scope (*environment model*).

These choices—verification scope, loop bounds, and environment model—are encoded in a *unit proof*, shown on the right of Listing 5.1. For example, Program Lines 15–16 assert that `dst` points to a valid allocated object and that the object is large enough for the indexed write; the unit proof then determines the scope, bounds, and assumptions under which BMC checks those assertions.

5.3.3 Verification Choice Fidelity

The guarantees afforded by BMC depend on the choices encoded in a unit proof. For example, on Line 13 of Listing 5.1, setting the bound for the program loop to 1 may suffice to check that `dst` is valid, but not that *all* indexed accesses are valid. Similarly, the overly-constrained model `get_record_count_m1` (Unit Proof Lines 4–8) masks the violation, whereas `get_record_count_m2` exposes it.

We define *verification choice fidelity (VCF)* as the degree to which these choices reflect the behavior of the real system. Formally, we characterize fidelity as $VCF = \langle S_{cf}, B_f, E_f \rangle$, where S_{cf} denotes verification-scope fidelity, B_f denotes loop-bound fidelity, and E_f denotes environment-model fidelity. This notion is related to model fidelity in systems modeling [195] and execution fidelity in firmware rehosting [182, 196] but focuses on choices that affect bounded model checking. Higher-fidelity choices preserve implementation behavior and can support stronger guarantees. Lower-fidelity choices simplify semantics, reducing development cost.

Many existing approaches to component-level BMC are *fidelity-oriented*. They rely on experts to encode detailed knowledge of the system [122], infer artifacts from specifications [128] or unit tests [129], or refine artifacts manually using verification results [123]. Such proofs can provide strong guarantees, but require deep knowledge of the component and its role in the surrounding program. It is therefore hard to create them automatically.

Other approaches rely on automatable choices, such as fixed function-level scopes [149, 197], uniform loop bounds [198], or environment models inferred by program analysis [160, 197]. However, these methods focus on restricted property sets, limiting their use for memory-safety verification; or require expert guidance to apply them in real software. This gap motivates methods that derive verification choices automatically while preserving the safety-relevant behavior needed for useful memory-safety verification guarantees.

5.4 Problem Statement

Our goal is to automate component-level memory-safety verification through the creation of unit proofs. This requires identifying verification choices that provide meaningful memory-safety guarantees while keeping verification tractable.

We formalize the problem as follows. Given a software system S, a component entry point C_{e}, a set of memory-safety properties Q, and a resource budget R, automatically construct a unit proof $U(V)$ that encodes the verification choices

$$V = (S_c, B, E),$$

where $S_c \subseteq S$ is a connected set of functions rooted in C_{e}, B maps each loop reachable in S_c to a maximum unrolling bound, and E is an environment model that defines assumptions for functions outside S_c. The resulting proof should allow BMC to verify Q conclusively within R, while ensuring the result reflects the memory safety of the verified functions S_c in the original system S. In Listing 5.1, this means constructing the unit proof on the right so that BMC exposes the memory-safety error on the left.

No existing work solves this problem automatically. Prior work on memory-safety verification either requires experts to define these choices [122, 128, 129], or uses fixed scope, loop bounds and program-analysis-inferred environment models [149, 160, 197] that require expert adaptation to real software. In this work, we automatically derive verification choices and construct executable unit proofs for real C software.

5.4.1 Success Criteria

A unit proof should support useful memory-safety guarantees. Building on [123], we distinguish five requirements:

(1) Structural Validity: It should compile and verify the intended component identified by the entry point C_e. For example, the unit proof in Listing 5.1 must verify `process_record`.

(2) Conclusiveness [123]: It should produce a verification result within the resource budget R. The result may be either a proof that the target memory-safety properties Q hold, or a counterexample showing the states and execution trace that violate them.

(3) Verification Coverage [123]: It should verify as much reachable code in its verification scope as possible. We define *verification coverage* as the proportion of included lines that are exercised and checked by the unit proof. This metric is analogous to line coverage in unit testing and fuzzing.

(4) Result Validity: It should make verification choices that neither encode spurious violations nor mask feasible memory-safety violations within the checked scope, bounds, and assumptions. Reported violations should correspond to real memory-safety errors.

(5) Maintainability: Unit proofs are reusable and auditable artifacts. As a result, their format should be familiar to software engineers so that the verification choices can be inspected and refined and the unit proofs maintained as the code evolves.

Automatically generating unit proofs that satisfy these requirements is non-trivial because the requirements can conflict. For example, a system may improve conclusiveness with shallow bounds or simple models, but reduce verification coverage or result validity. Conversely, techniques optimized for result validity may compromise coverage and tractability, or yield unit proofs too complex for engineering teams. Prior work [123] introduced conclusiveness and verification coverage to assess unit proof completeness. We add structural validity, result validity, and maintainability to capture risks introduced by automation: a generated unit proof may verify the wrong entry point, encode choices that produce invalid results, or become too complex for maintainers to inspect and refine.

5.4.2 System and Threat Model

System Model: We consider software systems S written in the C language, which is often used for embedded software development. We focus on components that process untrusted inputs from external interfaces, *e.g.,* network channels, since these interfaces form primary attack surfaces. We exclude programs in which memory references can be shared across processes (threads or tasks) and that are prone to race conditions, because they violate the linear execution assumptions of standard bounded model checking tools.

Threat Model: We consider attackers who can supply inputs through external interfaces and thereby trigger violations of the target memory-safety properties Q. Such inputs may flow directly to memory-safety sinks or indirectly influence control/data flow so that the component reaches an unsafe memory state.

5.5 AutoSOUP Design and Implementation

We designed AutoSOUP to automate component-level memory-safety verification through the creation of unit proofs (Figure 5.2).

5.5.1 Key Ideas

AutoSOUP explores two ideas: safety-oriented derivation of verification choices (§5.5.1) and LLM-as-function-call automation (§5.5.1).

Memory-Safety-Oriented Unit Proofs

We introduce *memory-safety-oriented unit proofs*: unit proofs whose verification choices $V = (S_c, B, E)$ are tailored to the instrumented memory-safety properties Q. Unlike fidelity-oriented unit proofs, which choose V to approximate the program's behavior, memory-safety-oriented unit proofs choose V to preserve the behavior needed to prove or refute Q within the resource budget R. This gives a safety-oriented interpretation of each choice: S_c should include code that contributes checkable memory-safety behavior; B should be large enough

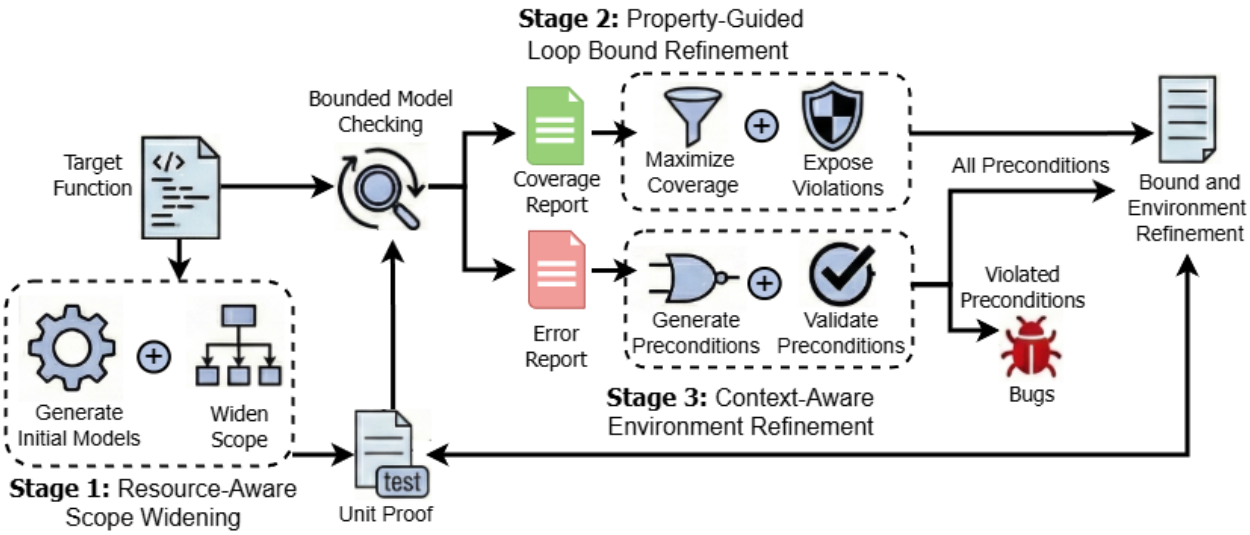

Figure 5.2. AutoSOUP system diagram. AutoSOUP connects safety-oriented unit-proof construction with LLM-as-function-call automation. Across all stages, deterministic workflows derive the verification choices $V = (S_c, B, E)$, delegate bounded semantic tasks to LLM agents, and validate returned artifacts before incorporating them into the unit proof.

to cover or expose property-relevant loop behavior; and E should expose possible violations while excluding infeasible ones.

AutoSOUP realizes this interpretation through three techniques.

1. Resource-Aware Scope Widening: Prior fidelity-oriented approaches rely on experts to select verification scopes [122, 198]. Automating this selection is difficult because it requires reasoning about both inter-function semantics and verification cost. However, prior work [123] suggests that memory-safety verification often does not require preserving all inter-function behavior.

We therefore treat scope selection as a resource-aware coverage problem rather than a full semantic-modeling problem. AutoSOUP widens S_c incrementally, using simple models for functions that remain outside the current scope. This lets the verifier check memory-safety properties in the included code without requiring precise models for every external callee. A widened scope is useful only if it increases the code and properties checked while remaining within the resource budget R. This reframing avoids the need to predict the globally best scope in advance: AutoSOUP starts with a small, tractable scope and expands only as needed.

2. Property-Guided Loop-Bound Refinement: Prior fidelity-oriented approaches rely on experts to define loop bounds [122] or loop invariants [199]. Other approaches use uniform bounds for all loops [198], which can be expensive and still miss memory-safety-relevant behavior. We instead treat loop-bound selection as a property-guided refinement problem. The intuition is that many memory-safety properties depend on local program states and are affected most directly by nearby loops. Thus, AutoSOUP refines B only for loops whose iterations affect the reachability or violation of instrumented memory-safety properties (*e.g.,* the loop on Line 13 of Listing 5.1).

3. Context-Aware Environment Refinement: Environment models create a different tension: permissive models expose possible memory-safety violations, but can also produce infeasible counterexamples; overly constrained models avoid spurious reports, but can mask real violations. We therefore extend counterexample-guided environment refinement [197] with context validation. AutoSOUP starts with permissive models that expose possible vi-

olations, infers preconditions that suppress infeasible violations, and then validates those preconditions against calling contexts in the original program. Preconditions satisfied by the surrounding program become explicit assumptions in E; violated preconditions identify feasible memory-safety errors. In Listing 5.1, this process distinguishes assumptions needed for `process_record` to be memory safe from assumptions violated by the actual implementation of `get_record_count()`.

Together, these techniques derive S_c, B, and E in a property-directed way. The resulting unit proofs $U(V)$ are safety-oriented because they preserve behavior relevant to the target safety properties rather than all program behavior. Although we focus on memory safety, the same formulation applies to safety properties expressible as Boolean predicates over program states.

Automating Safety-Oriented Unit Proofs

Automating safety-oriented unit proofs requires both project-specific reasoning and auditable verification choices. Program analysis can enforce syntactic and semantic constraints, but struggles with build configuration, local code idioms, and scalable semantic inference. LLMs can handle such project-specific reasoning, but unconstrained generation is unsuitable because invalid scopes, bounds, or environment assumptions can invalidate the resulting verification guarantees.

AutoSOUP balances reliability and generalizability with an *LLM-as-function-call* architecture. Deterministic algorithms control the proof-construction workflow and define the objective of each step. When a step requires semantic code understanding or project-specific adaptation, AutoSOUP delegates a bounded task to a tool-equipped LLM agent. Program-analysis modules then validate the returned artifact before it is incorporated into the unit proof.

5.5.2 Resource-Aware Scope Widening

Resource-aware scope widening derives S_c by incrementally adding semantically related code that may contain checkable memory-safety properties, retaining expansions only while

verification remains within the resource budget R. We use source files as the unit of expansion because related functions are often colocated, making file-level widening a coarse but useful approximation of semantic locality. The algorithm proceeds in three steps.

Step 1: Initialize the Scope, Bounds, and Input Model: AutoSOUP initializes S_c with the file containing C_e, sets all loop bounds to 1, and constructs a type-directed input model for C_e: primitive arguments receive unconstrained symbolic values, while pointer arguments receive valid allocated objects containing unconstrained values. An LLM agent synthesizes this input model and recovers the build configuration needed to compile the entry point's parent file, including headers, include paths, macros, and mandatory project configurations. AutoSOUP accepts the result only if deterministic checks confirm that the proof compiles, calls C_e, and introduces no preconditions beyond the intended type-directed initialization.

Step 2: Model External Calls: AutoSOUP identifies call edges that cross the current scope boundary and replaces their targets with simple type-based models. Primitive returns are modeled as unconstrained symbolic values, while pointer returns are modeled as valid allocated objects to avoid irrelevant invalid-pointer states that inflate the solver state space. An LLM agent synthesizes and integrates these models from the recovered call graph and return types; deterministic checks confirm that the resulting unit proof remains structurally valid.

Step 3: Widen the Scope: AutoSOUP checks the provisional unit proof against the configured time, memory, and file-depth budgets. If verification remains within R, it widens S_c by adding files that define previously modeled callees, selecting among ambiguous definitions by longest common path prefix to the in-scope caller. An LLM agent recovers build configuration for newly added files, and AutoSOUP repeats external-call modeling and scope widening until verification exceeds R or no additional files can be added.

Algorithm 1: Property-guided loop-bound and model refinement. Given a verification instance with initial loop bounds, refine the bounds, build configuration, and environment models to cover property-relevant code and expose loop-dependent memory-safety violations.

Input: Verification scope S_c, properties Q, loop bounds B, environment model E, resource budget R

Output: Refined loop-bound map B, refined environment model E

Function *PropertyGuidedRefinement*(S_c, Q, B, E, R)

    // Step 1: Cover property-relevant code
    $G \leftarrow$ UncoveredPropertyBlocks(S_c, Q, B, E)
    foreach $g \in G$ **do**
        $\rho \leftarrow$ ClassifyCoverageGap(g, S_c, B, E)
        $(B', E') \leftarrow$ RepairCoverageGap(g, ρ, B, E)
        if *ValidCoverageRefinement*(S_c, Q, B', E', R) **then**
            $(B, E) \leftarrow (B', E')$
    // Step 2: Expose loop-dependent property violations
    $L \leftarrow$ LoopsWithIncompleteUnwinding(S_c, B, E)
    foreach $\ell \in L$ **do**
        $P_\ell \leftarrow$ LoopDependentProperties(ℓ, Q)
        if $P_\ell \neq \emptyset$ **then**
            $n \leftarrow$ MinBoundToViolate(ℓ, P_ℓ)
            $B' \leftarrow B$; $B'[\ell] \leftarrow \max(B[\ell], n)$
            if *ValidBoundRefinement*(S_c, Q, B', E, R) **then**
                $B \leftarrow B'$
    return (B, E)

5.5.3 Property-Guided Loop & Model Refinement

Property-guided refinement derives the bounds B and model refinements E to exercise memory-safety-relevant behavior in S_c. It uses the instrumented properties Q as the oracle

for deciding which refinements matter. It honors the resource budget R by increasing bounds or refining models only to cover property-relevant code or expose violations. Our technique has two steps (algorithm 1).

Step 1: Cover Property-Relevant Code: A violation of $q \in Q$ can be exposed only if the line containing q is reached and checked. Coverage can be blocked by three factors: insufficient loop bounds, missing compile-time configuration, or external-call side effects not captured by return-value-only models.

AutoSOUP runs BMC in coverage mode, identifies uncovered property blocks, and uses an LLM agent to classify the blocking factor. It then applies the corresponding local repair: increase the relevant loop bound, adjust the configuration, or refine the external model to write unconstrained symbolic values through affected pointer arguments. The taxonomy and repair rules constrain the agent so that refinements remain local and aligned with the technique. A refinement is accepted only if the unit proof remains semantically valid, the target block becomes covered, and overall verification coverage does not decrease.

Step 2: Expose Loop-Dependent Property Violations: Covering a property does not imply that the current loop bounds can expose its violation. AutoSOUP inspects loops from the coverage report whose current bounds were insufficient for complete unwinding. For each such loop, it uses an LLM agent to determine whether the loop can affect violation of a nearby memory-safety property and, if so, to estimate the minimum bound likely to expose a violation. The prompt constrains this estimation using local program semantics, such as memory-region size, access stride, allocation constraints, and loop guards. A proposed bound is accepted only if it modifies the intended loop bound and preserves structural validity, conclusiveness, and verification coverage. If the bound causes verification to exceed R, AutoSOUP reports the bound but does not apply it.

5.5.4 Context-Aware Environment Model Refinement

Property-guided refinement in §5.5.3 maximizes exposure of violations of Q using permissive models E. However, violations found under permissive models may be infeasible in the broader system S. For example, the assertion on Program Line 15 in Listing 5.1 is vio-

lated if the input model provides a null pointer, even if the actual caller provides a statically allocated array. Our context-aware environment refinement separates infeasible violations (caused by overly permissive environment assumptions) from genuine memory-safety errors. This technique operates in two steps (algorithm 2).

Algorithm 2: Context-aware environment model refinement. The algorithm infers approximate preconditions for violated memory-safety properties, validates them against calling contexts, and reports caller-feasible violations as memory-safety errors.

Input: Software system S, component entry point C_{e}, Environment model E, Property violations Q_v

Output: Refined environment model E, memory-safety error set $\mathcal{M}$

Function *ContextAwareEnvRefinement*$(S, C_{\mathrm{e}}, E, Q_v)$
- $\mathcal{M} \leftarrow \emptyset$
- $\mathcal{W} \leftarrow$ `ParseViolationReport`(Q_v)
- `//` $\mathcal{W}$ `contains tuples` $(q, w(q))$
- **foreach** $(q, w(q)) \in \mathcal{W}$ **do**
  - $\phi \leftarrow$ `InferApproxPrecondition`$(E, q, w(q))$
  - $(\phi', \mathcal{B}) \leftarrow$ `ValidatePrecondition`$(S, C_{\mathrm{e}}, q, \phi)$
  - $E \leftarrow E \cup \{\phi'\}$
  - $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{B}$
- **return** $(E, \mathcal{M})$

Function *ValidatePrecondition*$(S, C_{\mathrm{e}}, q, \phi)$
- $\mathcal{B} \leftarrow \emptyset$; $\phi' \leftarrow \phi$
- $\mathcal{C} \leftarrow$ `CallsitesOf`(C_{e}, S)
- **foreach** $c \in \mathcal{C}$ **do**
  - $\psi_c \leftarrow$ `PathConstraints`(c, S)
  - **if** $\psi_c \not\models \phi'$ **then**
    - $\hat{\phi} \leftarrow$ `WeakenPrecondition`(ϕ', ψ_c)
    - **if** *SatisfiesProperty*$(\hat{\phi}, q)$ **then**
      - $\phi' \leftarrow \hat{\phi}$
    - **else**
      - $\mathcal{B} \leftarrow \mathcal{B} \cup \{(q, \psi_c, \phi')\}$
- **return** $(\phi', \mathcal{B})$

Step 1: Infer Approximate Weakest Preconditions: Following counter-example-guided environment refinement, AutoSOUP infers preconditions that suppress reported violations of memory-safety properties. These preconditions need not be logically weakest, since weakest-precondition inference is often computationally expensive [149]. Instead, they should be weak enough to preserve safe states while strong enough to suppress the target violation.

AutoSOUP parses the verification report to extract each violated property $q \in Q$, its location $loc(q)$, and its counterexample witness $w(q)$. For each tuple $(q, loc(q), w(q))$, an LLM agent infers a precondition that keeps $loc(q)$ covered but suppresses the violation of q. The prompt guides the agent to identify the violated condition, propagate it backward through local dataflow and path constraints, and stop at the external model or input responsible for the value. The agent can inspect witnesses, navigate code, and test candidate preconditions. A candidate is accepted only if it suppresses the target violation without reducing structural validity, conclusiveness, coverage, or the number of checked properties.

Step 2: Validate and Refine Against Calling Contexts: An inferred precondition may exclude feasible caller states that would not violate q, or it may reveal that the surrounding program can trigger the violation. AutoSOUP therefore validates each accepted precondition against calling contexts in S. Using a pre-indexed call graph, it identifies callsites of C_{e} and implementations of modeled functions. For each context, an LLM agent identifies path constraints reaching the callsite or implementation and checks whether those constraints imply the inferred precondition.

Validation has three outcomes. If a calling path violates the precondition but still satisfies q, the agent weakens and revalidates the precondition. If the precondition holds across the checked contexts, it becomes an explicit assumption in E. And of course, if a calling path violates the precondition and triggers q, AutoSOUP reports the path as a feasible memory-safety error in S. These assumptions make the generated unit proof auditable: the component is verified only under the preconditions recorded in its environment model.

5.5.5 Guarantees and Limitations

We discuss the guarantees and limitations of AutoSOUP.

Guarantees

Unit proofs generated by AutoSOUP provide bounded formal guarantees of memory safety, backed by BMC. They establish that *no execution from the component entry point violates a verified memory-safety property, provided the execution stays within the scope, loop bounds, and assumptions encoded in the unit proof.* This is the same form of guarantee provided by expert-written unit proofs [122]; its strength depends on the correctness and completeness of the encoded verification choices. AutoSOUP strengthens these choices by expanding safety-relevant scope, increasing bounds needed to expose property-relevant behavior, and making environment assumptions explicit.

Limitations

Three limitations constrain these guarantees. First, AutoSOUP uses LLMs for project-specific reasoning. To reduce unreliable outputs, deterministic workflows issue bounded tasks and validate outputs before incorporating them into the unit proof. As program analyses improve, these modules can be replaced with deterministic techniques. Second, AutoSOUP relies on static analysis to recover call graphs for scope widening and precondition validation. Imprecision, especially around indirect calls, may cause AutoSOUP to miss relevant call edges; engineers can reduce this risk by providing accurate application call graphs. Third, AutoSOUP widens scope at file granularity. Large files may exceed R and prevent wider scopes from being explored.

5.5.6 Implementation

We implement AutoSOUP in 15,463 lines of Python and 1,230 lines of prompts. Resource-aware scope widening uses 1,227 lines of Python and 292 prompt lines; property-guided loop-bound refinement uses 775 lines of Python and 316 prompt lines; and context-aware environment refinement uses 1,091 lines of Python and 479 prompt lines.

BMC Backend: AutoSOUP uses the ANSI-C Bounded Model Checker (CBMC) [141] as its verification backend. AutoSOUP can be used with most bounded model checkers that instrument the spatial and temporal memory-safety properties described in §2.2.

Program-Analysis Modules: AutoSOUP integrates components for deterministic program analysis. We use CBMC's `goto-instrument` to extract call-graph and symbol information from the compiled verification scope, which lets AutoSOUP check that the unit proof invokes the target entry point and identify undefined external callees during scope widening. We use libclang to locate function-pointer calls in the verification scope, allowing AutoSOUP to constrain them to selected models or concrete targets. We use cscope to index the software system S, recover project-level call relations, locate candidate function definitions, and identify calling contexts for validating inferred preconditions.

LLM-Driven Modules: AutoSOUP implements its LLM-driven modules using the OpenAI Python SDK [200] and LiteLLM [201]. The OpenAI SDK provides access to the GPT-family models used in our evaluation, while LiteLLM supports additional providers, including open-source and self-hosted models.

We expose three tools to the agents: *(i)* a *containerized terminal tool* lets agents inspect source files, build logs, and verification reports; *(ii)* a *cscope-based navigation tool* supports code search, definition lookup, and call-graph queries; and *(iii)* a *unit-proof validation tool* analyzes the compiled unit and verification report after each proposed change to confirm the proof compiles, calls the target entry point, satisfies the requested refinement task, and does not reduce line coverage or the number of covered or verified properties.

5.6 Evaluation

We structure our evaluation with four research questions:

- **RQ1:** Can AutoSOUP generate useful unit proofs for memory-safety verification?
- **RQ2:** Are memory-safety-oriented unit proofs effective at exposing memory-safety vulnerabilities?
- **RQ3:** How do AutoSOUP's safety-oriented techniques contribute to its performance and cost?

- **RQ4:** Do AutoSOUP-generated unit proofs differ from those of experts?

5.6.1 Evaluation Setup

Subjects: Following prior works on validating embedded software [123], we evaluate AutoSOUP using four widely-used open-source embedded operating systems: *Zephyr RTOS*, *RIOT OS*, *Contiki-NG* and *FreeRTOS*. These operating systems are substantial and include task management, IPC, networking, and storage subsystems. We considered including downstream embedded applications but their source code is often not publicly available [182].

Vulnerability Selection: To assess AutoSOUP's effectiveness to expose security vulnerabilities, we identify and recreate 60 known CVEs in selected subjects. To ensure reliable recreation, we extract CVEs affecting each software from the National Vulnerability Database and identify the first 20 CVEs where the vulnerable function still exists and the security advisory provided details of the vulnerability sink and the fixing commit. The FreeRTOS CVEs in our dataset did not provide the fixing commits and were excluded. We re-expose each CVE by applying a patch to reverse the changes in the identified fixing commit. For each CVE, we also record the vulnerability type, affected function, and sink location, to enable exposure detection.

Verification Targets: Component-level verification require entry points through which the components would be verified. We identify these entry points in three steps. First, we include the 57 functions affected by the 60 selected CVEs [1]. Second, to assess generalizability beyond known vulnerable functions, we randomly select an additional 100 functions across the four embedded OSes. First, we identify five modules in each operating system that handle untrusted data, such as bluetooth, network and USB modules. For each module, we perform an attack surface analysis to identify sources of untrusted data and the functions that process them. We select 5 functions per module, yielding an additional 25 functions per OS. Finally, we select 20 FreeRTOS functions with expert-written harnesses to enable head-to-head comparison in RQ4. Our component entry point set thus has a total of 177 entry points. The selected entry points span the parent modules described above.

[1] ↑3 functions contained 2 CVEs each

Baselines: We compare AutoSOUP against two categories of baselines. First, we compare against alternative methods for producing unit proofs, including expert-written FreeRTOS proofs and proofs generated by vanilla coding agents (GPT Codex). This comparison evaluates differences in verification choices and resulting outcomes. Second, we compare against memory-safety verification techniques, using Seeker [198], a recent open-source verifier for memory safety of open programs. Seeker is the closest prior system to our setting. We defer conceptual comparisons with other verification methods to §5.7.2.

Hardware and AI Models: We run experiments on dedicated servers (32 virtual CPUs, 188GB of RAM). We use gpt-5.3-codex through their API for our evaluation, as it ranked among the best AI models for coding [202] at time of study. We configure with the default temperature (1.0) and reasoning effort (high). To assess generalizability to open-source models, we compare against Minimax M2.5 and GLM-5, two leading open-source models for coding [202].

AutoSOUP Configuration: We evaluate two AutoSOUP configurations that differ in the maximum scope level used by resource-aware scope widening (§5.5.2). Scope-1 (*S1*) widens only to scope level of one (parent file containing entry point). Scope-2 (*S2*) uses depth two (parent file and all adjacent files). For each verification run, we use a 30-minute timeout, a practical budget under which most component-level verification tasks complete. We use the default configurations for the baseline techniques.

5.6.2 RQ1: Usefulness of AutoSOUP's Results

§5.4.1 identifies five properties that unit proofs must satisfy to support useful verification guarantees: structural validity, conclusiveness, verification coverage, result validity and maintainability. RQ1 evaluates the first three and last properties. We defer result validity to RQ2.

Method

We evaluate RQ1 on the full set of selected component entry points and use both AutoSOUP configurations. We run AutoSOUP-S1 on all verification targets and AutoSOUP-S2

only on targets with recreated CVEs because its wider scope increases generation and verification cost. We terminate generation runs that exceed 24 hours.

For each generated unit proof, we measure whether generation completes, whether the proof compiles, whether it is semantically valid, and how long verification takes. We then measure the verification outcome: the total lines of code in functions statically reachable from the unit proof, the proportion of those lines covered under the proof bounds and assumptions, the total number of instrumented memory-safety properties, and the proportion of those properties verified.

We also measure the cost and size of each unit proof. For cost, we record total generation time and API cost. For size, we record proof size in lines of code, including both harness function and all function models. Following prior work on unit testing [203, 204], we use unit proof size as a proxy for maintainability.

For comparison, we run Codex and Seeker on the verification targets with recreated vulnerabilities and report the corresponding measurements when available. For Codex, we use a prompt that states the goal of unit proof creation and the success criteria in §5.4.1, but gives no specific guidance on how to derive verification choices. This baseline tests whether a general-purpose AI coding agent can independently produce unit proofs that support useful memory-safety guarantees. For Seeker, we extract the compilation configurations from AutoSOUP-generated unit proofs and the scripts provided in the Seeker artifact to compile and verify the target file. To validate our setup, we sampled benchmarks from the Seeker codebase and confirmed that our instrumented programs matched the artifact versions and produced identical verification results.

Result

Unit proof validity: Table 5.1 compares the validity, verification outcomes, and generation cost of unit proofs produced by AutoSOUP and CodexUP. AutoSOUP produced substantially more valid unit proofs than Codex: 93%, 89.5% and 91.7% for scope levels 1 and 2 and the randomly selected targets, respectively, compared with 31.6% for CodexUP. In Seeker, of the 57 targets, 12 (21.1%) returned an error while 23 (40.4%) timed out.

Most Codex proofs were structurally invalid because Codex often failed to compile or verify the target, due to missing compilation, incomplete environment models or initially large loop bounds, and instead created simpler copies to verify. Seeker more often returned errors when it could not process the source file and timed out more often because it assigned a uniform bound of 20 to all loops and did not create valid environment models that reduce the state space. This shows that AutoSOUP's structured refinement and validation are necessary to produce unit proofs that compile, reach the target, and complete verification.

Table 5.1. Comparison of unit proofs produced by AutoSOUP at scope level 1 and 2 and by CodexUP. Random represents targets selected to assess generalizability. We report results from Seeker baseline in the prose. Below the double-line, we consider only the unit proofs from successful runs for each method.

Metric	**S1**	**S2**	**Codex**	**Random**
Num Targets	57	57	57	84 [2]
Struct. valid (%)	93%	89.5%	31.6%	91.7%
Verification completes (%)	96.5%	91.2%	70.2%	76.2%
Generation succeeds (%)	93.0%	78.9%	98.2%	75%
Verification time (s)	137.9	249.1	37.4	17.7
Avg compon. size (loc)	126.1	418.9	132.5	107.7
Avg covered size (loc)	108.8	147.4	57.8	88.8
Avg num. properties (#)	382.4	1187.2	468.3	243.2
Avg prop. verified (#)	369.5	1185.3	465.1	242.7
Avg reported errors (#)	4.7	3.2	1.6	1.2
Avg gen. time (min)	129.8	363.7	10.1	45.0
Avg API cost ($)	3.0	5.9	0.8	1.8
Avg proof size (loc)	34.1	40.1	50.7	27.3

The lower success rate of AutoSOUP-S2 relative to AutoSOUP-S1 is expected. Increasing the scope level adds more adjacent code to the verification target, which increases program size, state space, and verification time.

Verification outcomes and assurance: Among valid unit proofs, AutoSOUP, at scope levels 1 and 2, covered and verified 88.2% and 155% more lines of code compared to Codex' 57.8 lines of code. Data from Seeker is excluded as it does not provide coverage report. Table 5.1 also provides the number of total and verified properties as reported by the tool. However, because it counts the properties in non-covered code as verified because no violation was produced, Codex substantially lower coverage led to a higher number of reported verified properties. These results show that AutoSOUP also outperforms frontier coding agents in generating unit proofs that achieves better verification coverage and provides stronger memory-safety assurances.

The remaining uncovered code was primarily caused by statements that were not statically reachable from the generated unit proof, even when their enclosing functions were reachable. This effect becomes more pronounced as scope level increases and the proof includes functions from adjacent files. AutoSOUP nevertheless achieved higher coverage than CodexUP because its generated proofs are designed to admit all valid reachable states. By contrast, CodexUP often introduced restrictive assumptions in its unit proofs that excluded feasible states, reducing both code coverage and the corresponding number of properties checked.

Unit proof generation cost: Generating AutoSOUP unit proofs required 2.16 and 6.06 hours on average for scope levels 1 and 2, respectively, and cost $3 and $5.9 in API usage. These costs are higher than CodexUP, but they produce substantially more valid and useful proofs. They are also modest relative to prior human-driven unit proofing effort, where verifying 1,500 lines of code required about one person-month of work [122].

Figure 5.3 shows that the cost of AutoSOUP was distributed across the unit proofs. Overall, increasing the scope level from 1 to 2 increased the size of verified code, together with the development costs. Our data also showed these costs correlated closely with component size: larger component sizes took longer to verify, which increased the iterative-refinement-based development time.

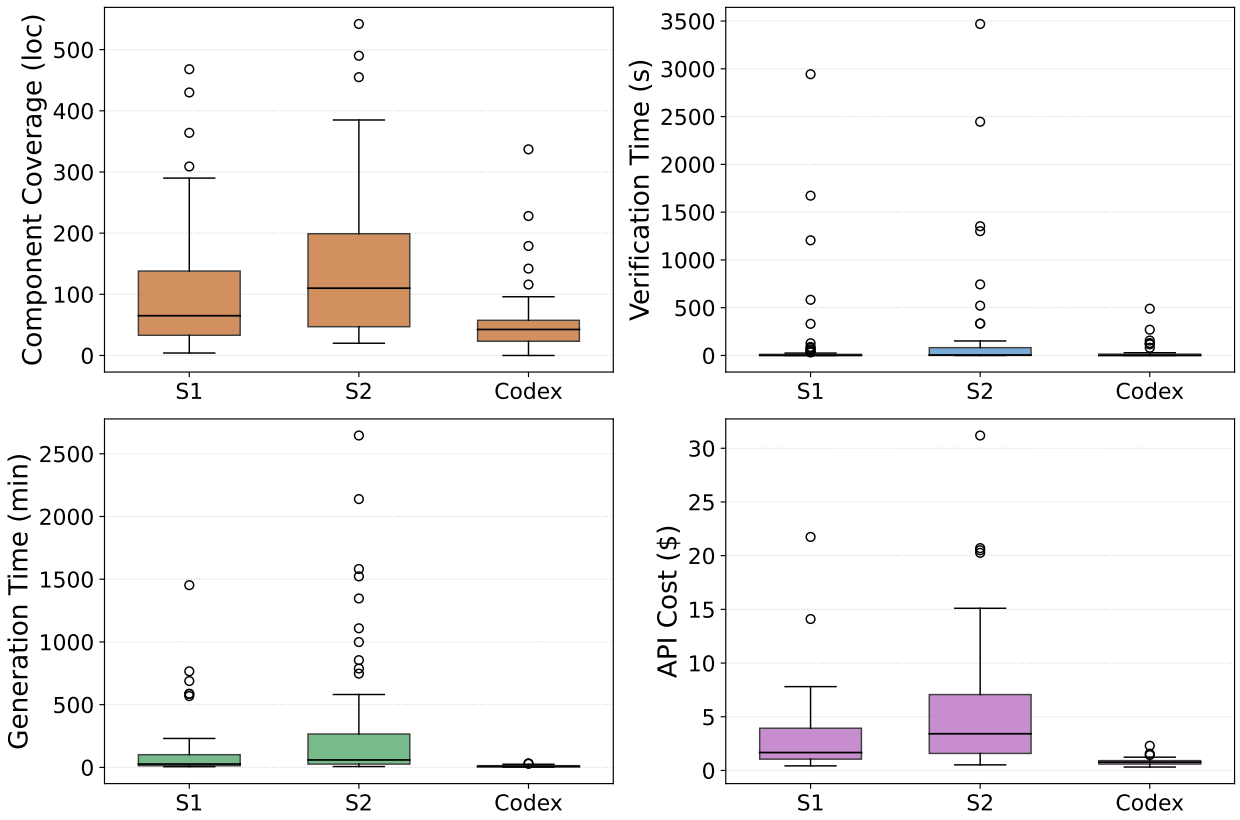

Figure 5.3. Distribution of program reachable LOC, verification time, development time, and API cost for the RQ1 unit proofs.

5.6.3 RQ2: Vulnerabilities Found by AutoSOUP

§5.4.1 requires unit proofs to produce valid memory-safety results: their verification outcomes should reflect genuine memory-safety vulnerabilities in the verified component. RQ2 evaluates whether unit proofs generated by AutoSOUP expose known memory-safety vulnerabilities during verification.

Method

We evaluate RQ2 on verification targets with recreated vulnerabilities. We compare AutoSOUP against the baselines using the proportion of vulnerabilities exposed and the root causes of any missed vulnerabilities.

Counting a vulnerability as exposed is non-trivial. During context-aware environment refinement (§5.5.4), AutoSOUP infers underapproximate preconditions that suppress reported property violations. A precondition inferred for one violation may also suppress other violations that share the same root cause. As a result, the violation location recorded by AutoSOUP may differ from the CVE sink location defined in §5.6.1, even when both correspond to the same vulnerability.

We therefore count a vulnerability as *exposed* if any of the following holds: (i) AutoSOUP reports a memory-safety error at the recorded CVE sink, and the inferred precondition can be violated at the component callsite; (ii) AutoSOUP reports a memory-safety error at a different sink, the inferred precondition can be violated at the callsite, and violating it would also trigger a property violation at the recorded CVE sink; or (iii) AutoSOUP reports a memory-safety error at the recorded CVE sink without inferring a precondition that suppresses the error.

We identify the first and third cases automatically by comparing recorded CVE sink locations with the memory-safety errors reported by AutoSOUP. We identify the second case manually by checking whether removing the inferred precondition produces a property violation at the recorded CVE sink. Finally, we compare the proportion of CVEs exposed by AutoSOUP, Codex, and Seeker, and manually categorize missed vulnerabilities to understand limitations of our techniques and those of the underlying bounded model checker.

Result

Exposure of known CVEs: Table 5.2 reports the proportion of recreated CVEs exposed by each method and summarizes the reasons for non-exposure. AutoSOUP-S1 and AutoSOUP-S2 exposed 66.7% and 65% of CVEs, respectively. In contrast, Codex and Seeker exposed only 28.3% and 41.7% [3]. These results show that AutoSOUP exposes substantially more known vulnerabilities than both a general-purpose coding agent and the prior verification baseline.

[3]↑For Seeker, due to limited information it provides, we counted any violation corresponding to the CVE sink line as exposure

Table 5.2. Exposure rate of known vulnerabilities. S1 and S2 represents AutoSOUP configured at scope levels one and two respectively.

Metric	S1	S2	Codex	Seeker
Number of CVEs	60	60	60	60
Number exposed	40	39	17	25
Unexploitable	6	5	1	-
Compilation error	3	4	6	-
Structural invalidity	0	0	2	-
Resource exhaustion	1	2	17	-
Limited scope	3	0	2	-
Limited loop unwinding	1	3	0	-
Inaccurate env. model	2	0	14	-
Unsupported sink	2	2	1	-
BMC field granularity	2	4	0	-

Root causes of non-exposure: We classify non-exposure into three categories. First, some unit proofs were invalid or inconclusive, preventing verification from producing a useful result. Second, some proofs used insufficient verification choices, such as scopes that excluded vulnerability-relevant paths, loop bounds that were too small, or environment assumptions that ruled out vulnerable states. Third, some cases were limited by the underlying bounded model checker, even when the relevant code and triggering conditions were present.

For AutoSOUP-S1, missed CVEs were primarily caused by limited scope, CBMC limitations, or vulnerabilities that were not triggerable from the project context. The scope-related misses were resolved by AutoSOUP-S2, showing that scope widening is important for exposing vulnerabilities whose triggers cross file boundaries.

The CBMC limitations appeared in two forms. First, CBMC reports a spatial memory-safety violation only when an access leaves the allocated object. However, for structs, AutoSOUP's property-guided loop-bound analysis computes the bound needed to overflow the

destination field, not necessarily the containing object. Thus, when a write overflows a struct field but remains within the same allocated struct object, CBMC does not report a violation. Second, 2 vulnerabilities was missed because they required semantics that CBMC did not model. One was a memory-leak, missed because the corresponding custom deallocator is not supported. The second required timer semantics for exposure.

In cases we tagged unexploitable, AutoSOUP reached the CVE sink and inferred the relevant precondition, but the precondition was valid in the component's calling context, either due to upstream constraints or default program configurations.

Codex missed CVEs mainly because its unit proofs were invalid, inconclusive, or over-constrained. Its ad hoc proof generation often produced scopes that did not support conclusive verification or assumptions that excluded vulnerable states. We could not investigate Seeker misses because no coverage report or unit proofs were produced.

Exposure of new vulnerabilities: During unit proof generation, AutoSOUP scope levels 1 and 2 reported 63 and 99 potential memory-safety violations where inferred preconditions were violated. After reviewing a subset of these reports, we confirmed 20 new externally triggerable vulnerabilities (Table 5.3). Nine of these are out-of-bound write vulnerabilities, with potentials to cause denial of service or arbitrary code execution. Another 7 are out-of-bound reads, which can potentially lead to information disclosure attacks. We reported all confirmed vulnerabilities to the corresponding maintainers. One has been fixed and assigned a CVE. Another 4 were fixed without CVE assignment because maintainers considered exploitation to require prior compromise of the downstream embedded application, which was outside their threat model. The remaining reports are still under investigation.

Example RIOT-OS Out-of-Bounds Write: A vulnerability in RIOT-OS `nanocoap.c` illustrates the defects discovered by AutoSOUP. AutoSOUP first generated a harness for the root function `coap_opt_put_uri_pathquery`, initializing `buf` and `string` as nondeterministic pointers with unconstrained sizes. Stage 2 refined the proof bounds and models until verification reached the vulnerable `memcpy` on Line 33, where a memcpy write violation was exposed. Finally, Stage 3 inferred a precondition on the input-buffer length that would eliminate the violation, but the validator traced the constrained string back to the nanocoap_sock_post public API and found no corresponding length check. AutoSOUP

therefore classified the precondition as violable in the real environment and reported the `memcpy` out-of-bounds write vulnerability.

Table 5.3. New vulnerabilities discovered using AutoSOUP. Each count is reported as *Total (Contiki-NG, Zephyr, RIOT)*.

Vulnerability Type	Count
Out-of-bound write	9 (5, 1, 2)
Out-of-bound read	7 (2, 3, 2)
Undefined shift behavior	2 (1, 1, 0)
Null pointer dereference	1 (1, 0, 2)
Total	20 (9, 5, 6)

5.6.4 RQ3: Ablation of AutoSOUP's Techniques

We ablate the techniques used to derive our safety-oriented unit proofs: verification scope, loop bounds, and environment models.

Method

During the RQ1 runs, we save a unit proof snapshot after each technique completes. For each snapshot, we also record the technique's execution time and API cost.

We use these snapshots to measure how each technique changes the unit proof. Specifically, we measure changes in verification scope, loop bounds, and environment models. We then execute each snapshot and measure the resulting verification coverage, the number of reachable memory-safety properties, and the proportion of those properties verified. This allows us to isolate how each verification choice affects verification outcomes and to estimate the marginal cost of each technique.

Finally, we compare generalizability to open-source models. We execute the 37 targets in RIOT and Contiki-NG using AutoSOUP configured at scope level 1 and equipped with

the selected open-source models, Minimax M2.5 and GLM-5. Similar to RQ1, we measure the validity of produced unit proofs, their verification outcomes and the generation cost.

Result

Contribution and cost of each technique: Table 5.4 summarizes how each AutoSOUP stage changes the unit proof, affects verification outcomes, and contributes to generation cost. Overall, the results show that the three techniques are complementary: each stage introduces a distinct class of verification choices and improves a different aspect of the final proof.

Table 5.4. Stage-wise contribution of AutoSOUP's scope, loop-bound, and environment-modeling stages to harness structure, verification behavior, development time, and API cost.

Metric	**Stage 1**	**Stage 2**	**Stage 3**
Harness Size (LOC)	6.613	7.795	21.64
Proof Size (LOC)	18.74	25.76	44.81
# Functions In Scope	17.78	18.84	17.34
# Custom Loop Bounds	0.004	0.909	0.842
# Variable Models	0.044	0.068	3.424
# Function Models	1.351	1.938	2.119
Avg Function Model Size (LOC)	2.954	3.602	5.402
Component Size (LOC)	209.8	220.8	196.3
Verification Coverage (LOC)	94.06	139.6	115.8
Total Properties	542.8	580.0	539.0
Verified Properties	412.3	412.8	531.6
Generation Time (min)	6.054	39.99	141.5
API Cost ($)	0.296	0.743	2.602

Stage 1 (resource-aware scope widening) determines the reachable functions included in the proof and introduces models for undefined functions. These choices remain largely unchanged in later stages. This stage is also the cheapest because it is implemented primarily using deterministic program analysis.

Stage 2 (property-guided loop refinement) increases the number of per-loop bounds. This expands the explored behavior of the component, which increases both verification coverage and the number of reachable memory-safety properties. At this stage, up to 71.04% of reachable properties on average are violated under the unconstrained environment, showing that loop unwinding exposes safety-relevant behaviors that must later be checked or constrained. This stage takes 39.99 minutes and costs $0.743 on average.

Stage 3 (context-aware environment refinement) primarily refines the variable and function models to eliminate infeasible executions while preserving safety-relevant behavior. This allows 98.63% of reachable properties to be verified and produces the environment assumptions under which the verification holds. It is also the most expensive because AutoSOUP infers and validate preconditions for reported property violations one by one.

Together, these results explain AutoSOUP's performance. Scope widening determines what code is analyzed, loop unwinding exposes safety-relevant behaviors within that code, and environment refinement separates feasible violations from behaviors ruled out by the component context.

Generalizability to open-source models: We also evaluate AutoSOUP with open-source models. Across 37 test cases, unit proof generation succeeded for 78.4% with GLM-5 and 48.6% with Minimax M2.5, compared to 93% with GPT-5.3-Codex. Their average costs were $4.9 and $0.7, respectively, compared to $3 for GPT-5.3-Codex. These results show that AutoSOUP can generalize to open-source models, but its performance depends on model capability. As open-source models improve, we expect corresponding gains in AutoSOUP's performance with them.

5.6.5 RQ4: AutoSOUP vs. Expert-Written Proofs

RQ4 compares unit proofs generated by AutoSOUP with existing FreeRTOS proofs. We restrict this analysis to FreeRTOS because it is the only evaluation subject with unit proofs developed by project maintainers.

Method

We first compare verification choices and outcomes quantitatively. For verification choices, we measure proof size, verification-scope size, distinct loop bounds, and the number of variable and function models. For outcomes, we measure verification time, component size, verification coverage, the number of verified properties, and the number of violated properties.

We then qualitatively analyze the environment models using the taxonomy from prior work [123]. Variable models fall into four categories: null-pointer preconditions (`p != NULL`), pointer-offset preconditions (`p2 = p1 + offset`), variable-constant preconditions (`var >= CONSTANT`), and variable-variable preconditions (`var1 >= var2`). Function models fall into three categories: Type 1 models with no preconditions, Type 2 models with preconditions only on return values, and Type 3 models with preconditions on inputs or global variables. We classify each model and compare the assumptions encoded by AutoSOUP-generated and expert-written proofs.

Finally, we conduct a case study of three randomly selected functions with varying proof sizes. We inspect their loop bounds and environment models to explain observed differences and their implications for verification.

Result

Quantitative comparisons: Figure 5.4 compares the verification choices and outcomes. Overall, AutoSOUP-generated unit proofs are smaller, use smaller verification scopes, include more variable models and fewer function models, and achieve comparable verification coverage. This difference stems from two design choices. First, AutoSOUP uses preconditions

to constrain environment variables, whereas FreeRTOS proofs often call project functions to initialize or constrain state, such as the proof for `vDHCPProcess` using `prvCreateDHCPSocket` to initialize DHCP sockets. Second, FreeRTOS proofs often replace same-file callees with function models and reuse shared function models across proofs. In contrast, AutoSOUP keeps same-file callees in scope and creates models only for functions outside the entry point's parent file, resulting in fewer function models.

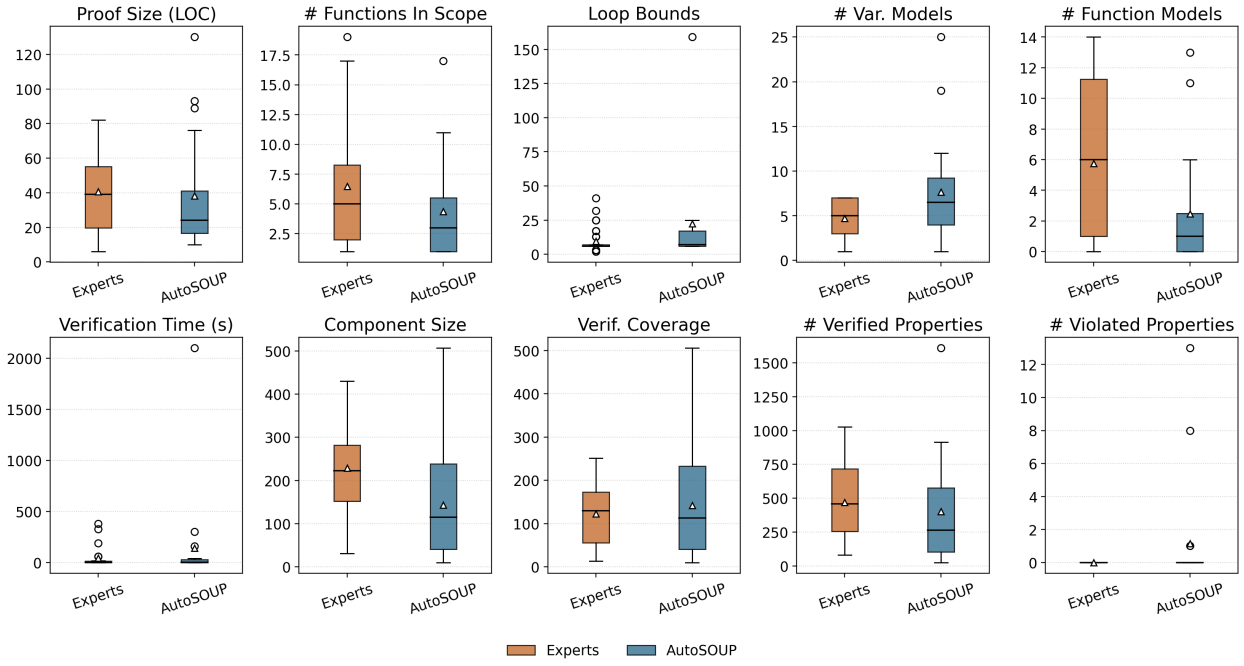

Figure 5.4. Comparing AutoSOUP's unit proofs with expert-written ones. Top measures verification choices. Bottom measures verification outcomes.

Qualitative comparisons: Table 5.5 shows the distribution of model categories. Although AutoSOUP introduces 90% more preconditions than the expert-written proofs, most are simple: non-null pointer constraints or fixed upper and lower bounds on environment variables needed for memory safety. Thus, AutoSOUP-generated environment models are comparable to expert-written models, making them auditable and maintainable. We also qualitatively compare specific generated and expert-written unit proofs when interpreting these results.

Table 5.5. Categorization of environment models from unit proofs. Data is aggregated from 20 unit proofs each. Top section represent variable model categories, while bottom section is for function models.

Model category	Expert-written	AutoSOUP
pointer-not-null	65	62
pointer-offset	2	1
variable-constant	38	70
variable-variable	3	8
Type 1 models	58	23
Type 2 models	18	19
Type 3 models	0	2

5.7 Discussion and Related Work

We discuss implications for practitioners and researchers and compare our work to related lines of research.

5.7.1 Discussion

Guarantees, Cost, and Practicality: AutoSOUP occupies a practical middle ground in the software-verification landscape. Deductive verification [185] can prove rich functional and memory-safety properties, but requires substantial specifications, annotations, and proof effort. Expert-written unit proofs for bounded model checking [122, 128] can provide useful memory-safety guarantees, but require experts to define scopes, loop bounds, inputs, and environment models. Automated analyzers such as Infer [205] operate at lower cost, but target narrower classes of properties than the bounded memory-safety guarantees studied in this paper. In contrast, AutoSOUP automates these choices and makes them explicit, while providing bounded memory-safety guarantees. RQ2 (§5.6.3) shows that this tradeoff is useful in practice: AutoSOUP exposes 66.7% of recreated CVEs and reports the assumptions under which each verified component is memory safe. It is also substantially cheaper than prior

industry experience where expert-written proofs take one engineer-month to cover 1500 lines of code. Although AutoSOUP does not replace stronger verification methods for safety-critical certification, it lowers the cost of memory-safety verification for broader software teams.

Integration into Developer Workflows: AutoSOUP makes component-level verification more practical for low-level software projects. RQ1 (§5.6.2) shows that it can generate unit proofs and verify components in realistic embedded software. This supports a practical *shift-security-left* workflow [206]: developers can obtain formal evidence about component memory safety during development, when defects are cheaper to diagnose and fix [207]. Our aeronautics industry partners noted that this capability is especially valuable in long development cycles, where products may take up to seven years to complete. In addition, by making environment assumptions explicit in the unit proof, AutoSOUP also exposes the interface contracts under which a component is memory safe. This will help developers identify and validate the caller-side obligations whose violation often leads to component-interface vulnerabilities [208].

Implications and Future Directions: AutoSOUP also illustrates a path for trustworthy AI-assisted software engineering. As developers increasingly use AI agents to generate and modify code, tools like AutoSOUP can provide machine-checkable evidence that generated components are memory safe under explicit assumptions. Its *LLM-as-function-call* architecture also allows model improvements to strengthen LLM-driven subtasks without weakening the deterministic orchestration and validation framework. This matters because formal guarantees depend on the assumptions encoded in the verification task [56], and automatically judging the correctness of these assumptions remains an open problem [188].

Looking forward, extending AutoSOUP from component-level memory-safety verification to broader project-level assurance requires three advances. First, loop unwinding should be combined or replaced with loop invariant generation [209] to expose vulnerabilities whose required bounds exceed the resource budget. Second, whole-project verification requires techniques that select effective component entry points, reduce bound or environment refinement cost with program analysis, and compose component-level results into project-level

guarantees [149, 210]. Third, generalizing beyond memory safety requires techniques that automatically infer system and security properties that can be verified using bounded model checking. Novel AI-driven techniques can help infer them from existing specifications, comments, and unit tests [211], then encode them as safety properties for verification.

5.7.2 Related Work

We situate our work within three lines of research.

(1) Automating Component-Level BMC: Prior work has explored automating bounded model checking through compositional verification [149, 197]. These techniques decompose programs along call-graph or control-flow edges, infer preconditions under which each fragment is safe, and discharge the resulting obligations using assume-guarantee reasoning [159]. However, it remains unclear whether these techniques scale to real-world systems with large components, diverse coding patterns, and complex inter-component relationships. Other work targets open programs, where components have undefined dependencies, and refines their environments using expert-provided precondition templates [157, 198]. Our evaluation shows that such templates can be imprecise and incomplete, compromising the resulting verification guarantees. In contrast, AutoSOUP automates industry-adopted component-level verification workflow and derives the scope, loop bounds, and environment models needed for practical memory-safety verification. The derived choices can also support compositional and open-program verification methods.

(2) LLMs and the Verification-Oracle Problem: Large language models are increasingly used to automate software-verification tasks, including tactic generation for deductive verification [212–214], precondition and loop-invariant inference for bounded model checking [209], and repair of verification failures [215]. In these settings, the LLM usually acts as the main driver and the task has a clear success oracle: the generated artifact is accepted if verification succeeds. AutoSOUP addresses a harder oracle problem because successful verification is not sufficient when unjustified choices of scope, bounds, or environment assumptions can make verification succeed without providing appropriate guarantees. Thus, AutoSOUP uses LLMs as assistants inside program-analysis-driven workflows: deterministic orchestrators is-

sue small tasks aligned with specific algorithms and validate each result before incorporating it into the unit proof. This architecture may also apply to other security and software-engineering tasks, such as fuzzing, where AI-generated harnesses can compile and achieve coverage yet still produce false crashes or invalid results [216–218].

(3) Security Vulnerability Detection: AutoSOUP complements static analysis [19, 219] and dynamic analysis [27, 55, 220] for vulnerability detection. Unlike these techniques, AutoSOUP produces auditable unit proofs that specify the scope, bounds, and assumptions under which a component is memory safe. These proofs can be rechecked to obtain bounded formal evidence of memory safety. Because AutoSOUP operates statically on source code, it is also well suited to embedded software, where dynamic analysis is often limited by emulation, rehosting, and peripheral-modeling challenges [196, 221–224].

5.8 Conclusion

AutoSOUP makes component-level bounded model checking more practical by automatically constructing unit proofs that verify a component's memory safety, while documenting the bounds and environment assumptions for the guarantees. It combines LLMs with a deterministic, incremental workflow to balance utility with reliability. Our evaluation on four substantial embedded RTOSes suggest automated harness generation as an incremental path toward security guarantees for real-world embedded software.

6. DISCUSSION

6.1 Revisiting the Thesis: From Ad Hoc Bug Finding to Systematic Assurance

§1.2 introduced the thesis statement of this dissertation: Memory-safety defects remain a critical and prevalent threat to software reliability and security, eliciting the need for higher-assurance validation methods. However, existing methods for validating memory safety either provide limited guarantees of memory safety (program analysis) or require costly and specialized expertise (formal verification). This problem is especially critical in embedded software systems where existing memory-safety validation methods are challenging to apply and errors are more costly to address after deployment. This dissertation addresses this gap by developing new systematic and automated techniques for high-assurance memory safety validation that do not require specialized expertise to adopt.

The results in Chapters 3–5 establish this thesis across a spectrum of assurance. Chapter 3 shows that dynamic analysis can be made more systematic by leveraging knowledge about recurring vulnerability patterns. EmNetTest replaces unconstrained input generation with field-level, systematic and ordered packet mutation and protocol-state exploration. It replicated 12 known vulnerabilities, discovered 7 new vulnerabilities across embedded network stacks and terminated, providing evidence that a defined class of packet processing behaviors, under the specified stride S and mutated field count N, is memory safe. Chapter 4 shows that a systematic, bottom-up, feedback-driven unit proofing approach can produce bounded guarantees of memory safety, expose memory-safety defects, and reduces the cost of creating unit proofs. Chapter 5 further advances systematic unit proofing by designing and automating techniques to infer verification scope, loop bounds and environment models for unit proofs. If soundly implemented, these techniques provide formal evidence that guarantees the absence of memory-safety vulnerabilities.

The techniques in this dissertation therefore provide assurances of memory safety that are stronger than those provided by ad hoc program analysis techniques from prior work [19–22]. They are also more practical than full formal verification because they substantially reduce the amount of manual effort and specialized expertise required by prior work [25, 56–58]. However, their guarantees are bounded by design: EmNetTest will not expose

vulnerabilities that require behaviors not covered under the specified stride and mutated field count. Similarly, the systematic unit proofing process and AutoSOUP do not expose vulnerabilities beyond the verification scope and bounds encoded in the proof harnesses or vulnerabilities that require higher-fidelity behavioral modeling. Their guarantees are also limited by the soundness of their implementation. However, our results show that despite these limitations, these techniques scale to real embedded software systems and are effective in exposing known and new exploitable memory-safety defects in analyzed software.

6.2 Full Project-Level Software Verification: Where Are We?

Systematic unit proofing and AutoSOUP both advance the practicality of software verification towards integration in software development workflows. However, by design, they only provide component-level assurances of memory safety and require a defined entry point and scope that remain tractable under verification. A successful unit proof only establishes that the verified component satisfies the instrumented memory-safety properties under the environment assumptions encoded in the unit proof but it does not guarantee that indeed, the component is memory safe when called from the parent project. While a component-level approach makes verification more practical, full project-level verification still requires techniques that can guarantee memory safety across the entire project when isolated and individually-verified components are composed together.

The remaining gap is primarily about project decomposition and proof composition. A project-level verifier must first identify verification units that are small enough for automated reasoning and large enough to capture safety-relevant interactions. After verifying each component, it must also connect the environment assumptions inferred in one proof to guarantees about callers, initialization code, configuration logic, hardware abstractions, or other verified components. It also requires new techniques that provide better support for unbounded loops and recursion so that the resulting verification guarantees remain unbounded. Prior work on compositional verification [149, 160, 210], assume-guarantee reasoning [157–159] and loop invariant generation [199, 209] provides a starting point but it remains unclear how they can be stitched together to provide practical verification for real-world software systems.

Beyond memory-safety verification, automated project-level verification should also provide guarantees about other correctness and security properties. Memory safety was a productive starting point for this dissertation because many memory-safety properties, such as invalid reads, invalid writes, null pointer dereferencing, are program-agnostic, can be specified as code-level assertions and are automatically instrumented by existing tools. Some other correctness and security properties, such as the correctness of data manipulating programs, can also be specified as code-level assertions but they require program-specific knowledge and specifications. In such cases, the verification techniques provided in this dissertation can help but they require new automated interventions to recover and specify these properties [211, 225]. Other properties require richer specifications or higher-fidelity environment models to verify. For example, protocol correctness requires semantic reasoning over state-machine properties[61, 226]. Concurrency correctness requires reasoning about interleavings [227, 228]. Functional correctness requires a specification of intended behavior [25, 185]. While verification of these properties remains beyond the scope of this dissertation, the systematization principles and property-guided verification choices developed here can inform future techniques for verifying them.

6.3 Generalizing Beyond Memory Safety and Embedded Software

The techniques developed in this dissertation can support properties beyond memory safety, although the required specifications differ. Systematic dynamic testing can target any property whose violation can be observed through a sanitizer or another executable runtime checker [49]. Similarly, systematic unit proofing and AutoSOUP can verify safety properties that can be expressed as Boolean predicates over program state and encoded as assertions. Memory-safety assertions are automatically instrumented by existing verification tools [141], whereas program-specific correctness and security properties must first be specified. Generalizing unit proofing to these properties therefore requires either developer-provided specifications or techniques that can automatically infer and validate candidate assertions.

The unit-proofing techniques also do not depend on an embedded-operating-system-specific architecture. They are practical for the systems studied because their modular components can be scoped into tractable verification units and most selected components do not involve pervasive shared-state concurrency that may modify relevant program state during verification. Modularity is a fundamental software-engineering principle, suggesting that these methods may also apply to libraries, drivers, middleware, and other componentized software. Systems with tightly coupled components or extensive concurrency may, however, require richer decomposition, environment modeling, and compositional reasoning, which can be adapted from prior work [210, 228]. Evaluating these conditions across other classes of software remains an important direction for future work.

Finally, the dissertation focuses on C and C++ because memory-safety defects are especially prevalent in these languages [29]. Memory-safe languages such as Rust reduce this risk, but Rust programs may still contain `unsafe` blocks or interact with memory-unsafe C libraries through foreign-function interfaces [13, 129]. The testing and verification methods developed here can be applied to these unsafe regions and integration boundaries. More broadly, unit proofing should extend to languages supported by bounded model checkers, including Rust through Kani and Java through JBMC [142, 150]. Such extensions would require adapting proof-harness generation, environment modeling, and completeness criteria to each language and verifier, while preserving the underlying systematic approach.

6.4 Systematic Assurance in the Era of AI-Generated Software

Recent frontier coding models, including Claude Fable 5 [229] and GPT-5.6 [230], can autonomously write software, inspect large codebases, and discover complex defects. These capabilities do not eliminate the need for the techniques developed in this dissertation. AI agents are non-deterministic [231, 232], and a successful agent run does not establish that defects are absent from the resulting software. The evaluation in Chapter 5 illustrates this distinction: the general-purpose Codex baseline underperformed AutoSOUP both in terms of the number of valid unit proofs produced and the number of security vulnerabilities discovered [28]. Frontier agents can assist in vulnerability discovery and other verification tasks,

but stronger assurance still requires systematic techniques whose results and assumptions can be independently checked.

The growing use of AI for both software development and vulnerability discovery makes such assurance increasingly important. AI-generated code may increase the volume and speed at which software changes are produced, while the same capabilities can reduce the effort required to discover and exploit defects [233, 234]. This narrows the time available for maintainers to identify vulnerabilities before attackers do. The techniques in this dissertation are designed for incremental use within software-development workflows: systematic tests and unit proofs can be applied to individual components before release and rerun as those components change. They can therefore provide earlier evidence of memory safety and reduce reliance on post-release vulnerability discovery and remediation.

AI agents and systematic verification are most effective when combined. AutoSOUP can be packaged as a tool that coding agents invoke after generating or modifying code. An agent could run the generated unit proof, use reported counterexamples to revise its implementation, and repeat the process until the required verification criteria are satisfied. In this workflow, the agent contributes code generation, semantic interpretation, and repair, while deterministic program analyses and bounded model checking provide checkable evidence about the resulting code. This integration would extend the neuro-symbolic approach introduced in Chapter 5 from proof generation to a development process in which coding agents verify their changes before they are accepted or released.

7. CONCLUSION

This dissertation contributes toward the need for high assurance and practical memory safety validation techniques in software engineering workflows. Prior validation techniques either provide limited memory safety assurance (static analysis and fuzzing) or are too costly to adopt (formal verification). This need is more acute in embedded software systems where failures and cyberattacks have safety-critical consequences and errors are more costly to fix after deployment.

This dissertation closes this gap by designing systematic and automated techniques that deliver stronger memory safety guarantees, are effective in exposing vulnerabilities in embedded software, and remain practical to adopt. Chapter 3 showed that replacing random state exploration and input generation methods with systematic algorithms can lead to stronger memory safety guarantees in embedded network stacks: EmNetTest replicated 12 known vulnerabilities and discovered 7 new ones in under three hours per target, while random fuzzing with the same seeds found none within 24 hours. Chapter 4 then showed that stronger assurance through bounded model checking can be made practical by systematizing proof-harness development workflow: systematically developed unit proofs exposed 74% of studied memory-safety vulnerabilities without prior semantic knowledge, performed better than expert-developed proofs, and took only 72 minutes to verify 185 lines of C code. Chapter 5 advanced this result by introducing principled techniques for selecting verification choices for effective verification and automating the workflow introduced in Chapter 4. Across three major embedded operating systems, AutoSOUP verified up to 93% of components, exposed 66.7% of recreated CVEs, and identified 20 new vulnerabilities.

Taken together, these results provide evidence that stronger memory safety assurance can become achievable and affordable in practice using systematic and automated techniques. This will enable software engineering teams to validate the memory safety of software systems during development, produce justifiable evidence of memory safety guarantees, and deploy secure and vulnerability-free products to their end-users.

REFERENCES

[1] Alex Rebert, Chandler Carruth, Jen Engel, and Andy Qin. "Safer with google: Advancing memory safety," Google Online Security Blog. (), [Online]. Available: https://security.googleblog.com/2024/10/safer-with-google-advancing-memory.html (visited on 03/08/2025).

[2] MSRC Team. "A proactive approach to more secure code MSRC blog microsoft security response center." (), [Online]. Available: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ (visited on 03/08/2025).

[3] "Project zero: 0day "in the wild"," Project Zero. (), [Online]. Available: https://googleprojectzero.blogspot.com/p/0day.html (visited on 03/08/2025).

[4] "OpenSSL 'heartbleed' vulnerability (CVE-2014-0160) CISA." (Oct. 5, 2016), [Online]. Available: https://www.cisa.gov/news-events/alerts/2014/04/08/openssl-heartbleed-vulnerability-cve-2014-0160 (visited on 04/15/2025).

[5] *Widespread IT Outage Due to CrowdStrike Update CISA*, en, Aug. 2024. [Online]. Available: https://www.cisa.gov/news-events/alerts/2024/07/19/widespread-it-outage-due-crowdstrike-update.

[6] "Channel file 291 incident RCA is available CrowdStrike," CrowdStrike.com. (), [Online]. Available: https://www.crowdstrike.com/en-us/blog/channel-file-291-rca-available/ (visited on 03/08/2025).

[7] L. K. Wee. "Here comes the wave of insurance claims for the CrowdStrike outage," Business Insider. (), [Online]. Available: https://www.businessinsider.com/businesses-claiming-losses-crowdstrike-outage-insurance-billions-losses-cyber-policies-2024-7 (visited on 03/08/2025).

[8] Bob Lord. "The urgent need for memory safety in software products CISA." (Sep. 20, 2023), [Online]. Available: https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products (visited on 03/08/2025).

[9] Alex Rebert, Ben Laurie, Murali Vijayaraghavan, and Alex Richardson. "Securing tomorrow's software: The need for memory safety standards," Google Online Security Blog. (), [Online]. Available: https://security.googleblog.com/2025/02/securing-tomorrows-software-need-for.html (visited on 03/08/2025).

[10] R. N. Watson *et al.*, "It is time to standardize principles and practices for software memory safety," *Commun. ACM*, vol. 68, no. 2, pp. 40–45, Jan. 22, 2025, ISSN: 0001-0782. DOI: 10.1145/3708553. [Online]. Available: https://dl.acm.org/doi/10.1145/3708553 (visited on 03/08/2025).

[11] P. H. Feiler, "Challenges in validating safety-critical embedded systems," *SAE International Journal of Aerospace*, vol. 3, no. 1, pp. 109–116, Nov. 10, 2009, Number: 2009-01-3284, ISSN: 1946-3855, 1946-3901. DOI: 10.4271/2009-01-3284. [Online]. Available: https://www.sae.org/publications/technical-papers/content/2009-01-3284/ (visited on 09/24/2024).

[12] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating systems for low-end devices in the internet of things: A survey," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 720–734, Oct. 2016, Conference Name: IEEE Internet of Things Journal, ISSN: 2327-4662. DOI: 10.1109/JIOT.2015.2505901.

[13] A. Sharma, S. Sharma, S. R. Tanksalkar, S. Torres-Arias, and A. Machiry, "Rust for embedded systems: Current state and open problems," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '24, New York, NY, USA: Association for Computing Machinery, Dec. 9, 2024, pp. 2296–2310, ISBN: 9798400706363. DOI: 10.1145/3658644.3690275. [Online]. Available: https://dl.acm.org/doi/10.1145/3658644.3690275 (visited on 03/08/2025).

[14] L. H. Newman, "An operating system bug exposes 200 million critical devices," *Wired*, Section: tags, ISSN: 1059-1028. [Online]. Available: https://www.wired.com/story/vxworks-vulnerabilities-urgent11/ (visited on 01/04/2023).

[15] "AMNESIA:33," Forescout. (), [Online]. Available: https://www.forescout.com/research-labs/amnesia33/ (visited on 01/04/2023).

[16] Forescout. "Project memoria," Forescout. (), [Online]. Available: https://www.forescout.com/research-labs/project-memoria/ (visited on 01/04/2023).

[17] R. Yu *et al.*, "Building embedded systems like it's 1996," arXiv, arXiv:2203.06834, Mar. 13, 2022, type: article. DOI: 10.48550/arXiv.2203.06834. arXiv: 2203.06834[cs]. [Online]. Available: http://arxiv.org/abs/2203.06834 (visited on 05/20/2022).

[18] A. Abbasi, J. Wetzels, T. Holz, and S. Etalle, "Challenges in designing exploit mitigations for deeply embedded systems," in *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, Jun. 2019, pp. 31–46. DOI: 10.1109/EuroSP.2019.00013.

[19] Q. A. Chen, Z. Qian, Y. J. Jia, Y. Shao, and Z. M. Mao, "Static detection of packet injection vulnerabilities: A case for identifying attacker-controlled implicit information leaks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15, New York, NY, USA: Association for Computing Machinery, Oct. 12, 2015, pp. 388–400, ISBN: 978-1-4503-3832-5. DOI: 10.1145/2810103.2813643. [Online]. Available: https://dl.acm.org/doi/10.1145/2810103.2813643 (visited on 04/09/2023).

[20] H. Zhang *et al.*, "Statically discovering high-order taint style vulnerabilities in OS kernels," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21, New York, NY, USA: Association for Computing Machinery, Nov. 13, 2021, pp. 811–824, ISBN: 978-1-4503-8454-4. DOI: 10.1145/3460120.3484798. [Online]. Available: https://dl.acm.org/doi/10.1145/3460120.3484798 (visited on 04/11/2023).

[21] V. J. Manès *et al.*, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[22] C. Brant *et al.*, "Challenges and opportunities for practical and effective dynamic information flow tracking," *ACM Computing Surveys*, vol. 55, no. 1, 17:1–17:33, Nov. 23, 2021, ISSN: 0360-0300. DOI: 10.1145/3483790. [Online]. Available: https://dl.acm.org/doi/10.1145/3483790 (visited on 03/22/2024).

[23] L. Szekeres, M. Payer, T. Wei, and D. Song, "SoK: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, ISSN: 1081-6011, May 2013, pp. 48–62. DOI: 10.1109/SP.2013.13. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6547101 (visited on 03/10/2025).

[24] J. Ganz and S. Peisert, "ASLR: How robust is the randomness?" In *2017 IEEE Cybersecurity Development (SecDev)*, Sep. 2017, pp. 34–41. DOI: 10.1109/SecDev.2017.19. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8077804 (visited on 03/10/2025).

[25] G. Klein *et al.*, "seL4: Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09, New York, NY, USA: Association for Computing Machinery, Oct. 11, 2009, pp. 207–220, ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. [Online]. Available: https://dl.acm.org/doi/10.1145/1629575.1629596 (visited on 09/28/2024).

[26] D. Kroening, P. Schrammel, and M. Tautschnig, *CBMC: The c bounded model checker*, Feb. 5, 2023. DOI: 10.48550/arXiv.2302.02384. arXiv: 2302.02384[cs]. [Online]. Available: http://arxiv.org/abs/2302.02384 (visited on 08/23/2024).

[27] P. C. Amusuo, R. A. C. Méndez, Z. Xu, A. Machiry, and J. C. Davis, "Systematically detecting packet validation vulnerabilities in embedded network stacks," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ISSN: 2643-1572, Sep. 2023, pp. 926–938. DOI: 10.1109/ASE56229.2023.00095. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10298483 (visited on 10/18/2024).

[28] P. C. Amusuo *et al.*, "AutoSOUP: Safety-oriented unit proof generation for component-level memory-safety verification," arXiv, Technical report arXiv:2605.10712, 2026. DOI: 10.48550/arXiv.2605.10712. [Online]. Available: https://arxiv.org/abs/2605.10712.

[29] P. C. van Oorschot, "Memory errors and memory safety: C as a case study," *IEEE Security & Privacy*, vol. 21, no. 2, pp. 70–76, Mar. 2023, ISSN: 1558-4046. DOI: 10.1109/MSEC.2023.3236542. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10102611 (visited on 04/06/2026).

[30] *Memory safety*, in *Wikipedia*, Page Version ID: 1342741521, Mar. 10, 2026. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Memory_safety&oldid=1342741521 (visited on 04/06/2026).

[31] S. Nagarakatte, "Full spatial and temporal memory safety for c," *IEEE Security & Privacy*, vol. 22, no. 4, pp. 30–39, Jul. 2024, ISSN: 1558-4046. DOI: 10.1109/MSEC.2024.3363142. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10439147 (visited on 04/06/2026).

[32] A. Azevedo de Amorim, C. Hriţcu, and B. C. Pierce, "The meaning of memory safety," in *Principles of Security and Trust*, L. Bauer and R. Küsters, Eds., Cham: Springer International Publishing, 2018, pp. 79–105, ISBN: 978-3-319-89722-6. DOI: 10.1007/978-3-319-89722-6_4.

[33] M. Hicks. "What is memory safety? - the PL enthusiast," The Programming Languages Enthusiast. (Jul. 21, 2014), [Online]. Available: http://www.pl-enthusiast.net/2014/07/21/memory-safety/ (visited on 04/06/2026).

[34] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1, 1990, ISSN: 0001-0782. DOI: 10.1145/96267.96279. [Online]. Available: https://dl.acm.org/doi/10.1145/96267.96279 (visited on 04/06/2026).

[35] Aleph One. "Smashing the stack for fun and profit." (), [Online]. Available: https://archives.phrack.org/issues/49/14.txt (visited on 03/08/2025).

[36] M. Barr. "2005 Camry L4 Software Analysis (Bookout v. Toyota)," Michael Barr trial slides. (Nov. 6, 2013), [Online]. Available: https://www.autosafety.org/wp-content/uploads/import/Bookout%20PDF.pdf (visited on 04/06/2026).

[37] CrowdStrike, "External Technical Root Cause Analysis – Channel File 291," CrowdStrike, Aug. 6, 2024. [Online]. Available: https://www.crowdstrike.com/content/dam/crowdstrike/www/en-us/wp/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf (visited on 04/06/2026).

[38] J. T. F. T. Initiative, "Managing information security risk: Organization, mission, and information system view," National Institute of Standards and Technology, NIST Special Publication (SP) 800-39, Mar. 1, 2011. DOI: 10.6028/NIST.SP.800-39. [Online]. Available: https://csrc.nist.gov/pubs/sp/800/39/final (visited on 04/06/2026).

[39] E. A. Lee, "Embedded software," in *Advances in Computers*, vol. 56, Elsevier, 2002, pp. 55–95, ISBN: 978-0-12-012156-4. DOI: 10.1016/S0065-2458(02)80004-3. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0065245802800043 (visited on 07/13/2021).

[40] K. Chen *et al.*, "Internet-of-things security and vulnerabilities: Taxonomy, challenges, and practice," *Journal of Hardware and Systems Security*, vol. 2, no. 2, pp. 97–110, Jun. 2018, ISSN: 2509-3428, 2509-3436. DOI: 10.1007/s41635-017-0029-7. [Online]. Available: http://link.springer.com/10.1007/s41635-017-0029-7 (visited on 07/14/2021).

[41] "FreeRTOS™ - FreeRTOS™." (), [Online]. Available: https://freertos.org (visited on 09/29/2024).

[42] "Zephyr project documentation — zephyr project documentation." (), [Online]. Available: https://docs.zephyrproject.org/latest/index.html (visited on 12/03/2024).

[43] Wind River. "VxWorks RTOS Real-Time Operating System Wind River," Wind River. (2026), [Online]. Available: https://www.windriver.com/products/embedded/vxworks (visited on 04/06/2026).

[44] lwIP. "lwIP: Overview." (), [Online]. Available: https://www.nongnu.org/lwip/2_1_x/index.html (visited on 04/07/2023).

[45] A. Stanoev. "Contiki-NG: The OS for next generation IoT devices," GitHub. (), [Online]. Available: https://github.com/contiki-ng/contiki-ng/wiki/Home (visited on 04/07/2023).

[46] M. Shen, A. Pillai, B. A. Yuan, J. C. Davis, and A. Machiry, *An empirical study on the use of static analysis tools in open source embedded software*, Sep. 30, 2023. DOI: 10.48550/arXiv.2310.00205. arXiv: 2310.00205[cs]. [Online]. Available: http://arxiv.org/abs/2310.00205 (visited on 03/08/2025).

[47] S. Ma, M. Jiao, S. Zhang, W. Zhao, and D. W. Wang, "Practical null pointer dereference detection via value-dependence analysis," in *2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Nov. 2015, pp. 70–77. DOI: 10.1109/ISSREW.2015.7392049. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7392049 (visited on 03/13/2025).

[48] H. Yan, Y. Sui, S. Chen, and J. Xue, "Machine-learning-guided typestate analysis for static use-after-free detection," in *Proceedings of the 33rd Annual Computer Security Applications Conference*, ser. ACSAC '17, New York, NY, USA: Association for Computing Machinery, Dec. 4, 2017, pp. 42–54, ISBN: 978-1-4503-5345-8. DOI: 10.1145/3134600.3134620. [Online]. Available: https://dl.acm.org/doi/10.1145/3134600.3134620 (visited on 04/11/2023).

[49] K. Serebryany and D. Bruening, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX Association, 2012, pp. 309–318.

[50] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A greybox fuzzer for network protocols," in *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, ISSN: 2159-4848, Oct. 2020, pp. 460–465. DOI: 10.1109/ICST46399.2020.00062.

[51] R. Natella, "StateAFL: Greybox fuzzing for stateful network servers," *Empirical Software Engineering*, vol. 27, no. 7, Dec. 1, 2022, ISSN: 1382-3256. DOI: 10.1007/s10664-022-10233-3. [Online]. Available: https://doi.org/10.1007/s10664-022-10233-3 (visited on 08/14/2023).

[52] A. Andronidis and C. Cadar, "SnapFuzz: An efficient fuzzing framework for network applications," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, Jul. 18, 2022, pp. 340–351. DOI: 10.1145/3533767.3534376. arXiv: 2201.04048[cs]. [Online]. Available: http://arxiv.org/abs/2201.04048 (visited on 01/06/2023).

[53] Y.-H. Zou, J.-J. Bai, J. Zhou, J. Tan, C. Qin, and S.-M. Hu, "{TCP-fuzz}: Detecting memory and semantic bugs in {TCP} stacks with fuzzing," presented at the 2021 USENIX Annual Technical Conference (USENIX ATC 21), 2021, pp. 489–502, ISBN: 978-1-939133-23-6. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/zou (visited on 10/19/2022).

[54] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices," in *Proceedings 2018 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2018, ISBN: 978-1-891562-49-5. DOI: 10.14722/ndss.2018.23166. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-4_Muench_paper.pdf (visited on 03/10/2025).

[55] J. Yun, F. Rustamov, J. Kim, and Y. Shin, "Fuzzing of embedded systems: A survey," *ACM Computing Surveys*, vol. 55, no. 7, pp. 1–33, Jul. 31, 2023, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3538644. [Online]. Available: https://dl.acm.org/doi/10.1145/3538644 (visited on 03/10/2025).

[56] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy, "An empirical study on the correctness of formally verified distributed systems," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17, New York, NY, USA: Association for Computing Machinery, Apr. 23, 2017, pp. 328–343, ISBN: 978-1-4503-4938-3. DOI: 10.1145/3064176.3064183. [Online]. Available: https://dl.acm.org/doi/10.1145/3064176.3064183 (visited on 09/28/2024).

[57] M. Luckcuck, M. Farrell, L. A. Dennis, C. Dixon, and M. Fisher, "Formal specification and verification of autonomous robotic systems: A survey," *ACM Computing Surveys*, vol. 52, no. 5, pp. 1–41, Sep. 30, 2020, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3342355. [Online]. Available: https://dl.acm.org/doi/10.1145/3342355 (visited on 09/29/2024).

[58] L. Huang, S. Ebersold, A. Kogtenkov, B. Meyer, and Y. Liu, *Lessons from formally verified deployed software systems (extended version)*, Mar. 28, 2024. DOI: 10.48550/arXiv.2301.02206. arXiv: 2301.02206[cs]. [Online]. Available: http://arxiv.org/abs/2301.02206 (visited on 03/09/2025).

[59] A. Lattuada *et al.*, "Verus: Verifying rust programs using linear ghost types," *Proc. ACM Program. Lang.*, vol. 7, 85:286–85:315, OOPSLA1 Apr. 6, 2023. DOI: 10.1145/3586037. [Online]. Available: https://dl.acm.org/doi/10.1145/3586037 (visited on 10/01/2024).

[60] FreeRTOS. "FreeRTOS - market leading RTOS (real time operating system) for embedded systems with internet of things extensions," FreeRTOS. (), [Online]. Available: https://www.freertos.org/index.html (visited on 04/07/2023).

[61] L. Lockefeer, D. M. Williams, and W. Fokkink, "Formal specification and verification of TCP extended with the window scale option," *Science of Computer Programming*, Formal Methods for Industrial Critical Systems (FMICS'2014), vol. 118, pp. 3–23, Mar. 1, 2016, ISSN: 0167-6423. DOI: 10.1016/j.scico.2015.08.005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642315001835 (visited on 04/10/2023).

[62] M. A. S. Smith, "Formal verification of TCP and t/TCP,"

[63] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations,"

[64] A. Zaostrovnykh *et al.*, "Verifying software network functions with no verification expertise," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19, New York, NY, USA: Association for Computing Machinery, Oct. 27, 2019, pp. 275–290, ISBN: 978-1-4503-6873-5. DOI: 10.1145/3341301.3359647. [Online]. Available: https://dl.acm.org/doi/10.1145/3341301.3359647 (visited on 04/07/2023).

[65] K. Zhang, D. Zhuo, A. Akella, and A. K. X. Wang, "Automated verification of customizable middlebox properties with gravel,"

[66] S. Pirelli, A. Valentukonytė, K. Argyraki, and G. Candea, "Automated verification of network function binaries," presented at the 19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), 2022, pp. 585–600, ISBN: 978-1-939133-27-4. [Online]. Available: https://www.usenix.org/conference/nsdi22/presentation/pirelli (visited on 04/07/2023).

[67] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi, "Finding protocol manipulation attacks," in *Proceedings of the ACM SIGCOMM 2011 conference*, ser. SIGCOMM '11, New York, NY, USA: Association for Computing Machinery, Aug. 15, 2011, pp. 26–37, ISBN: 978-1-4503-0797-0. DOI: 10.1145/2018436.2018440. [Online]. Available: https://dl.acm.org/doi/10.1145/2018436.2018440 (visited on 04/16/2023).

[68] R. Chang, G. Jiang, F. Ivancic, S. Sankaranarayanan, and V. Shmatikov, "Inputs of coma: Static detection of denial-of-service vulnerabilities," in *2009 22nd IEEE Computer Security Foundations Symposium*, ISSN: 2377-5459, Jul. 2009, pp. 186–199. DOI: 10.1109/CSF.2009.13.

[69] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein, "Analyzing protocol implementations for interoperability," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15, USA: USENIX Association, May 4, 2015, pp. 485–498, ISBN: 978-1-931971-21-8. (visited on 08/14/2023).

[70] C. Poncelet, K. Sagonas, and N. Tsiftes, "So many fuzzers, so little time : Experience from evaluating fuzzers on the contiki-NG network (hay)stack," presented at the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022. [Online]. Available: http://urn.kb.se/resolve?urn=urn:nbn:se:ri:diva-61138 (visited on 01/06/2023).

[71] D. Anandayuvaraj, P. Thulluri, J. Figueroa, H. Shandilya, and J. C. Davis, *Towards a failure-aware SDLC for internet of things*, 2022. [Online]. Available: https://arxiv.org/abs/2206.13562.

[72] D. A. Norman, "Commentary: Human error and the design of computer systems," *Communications of the ACM*, vol. 33, no. 1, pp. 4–7, 1990, Publisher: Association for Computing Machinery, Inc.

[73] C. Johnson, "Software support for incident reporting systems in safety-critical applications," in *Computer Safety, Reliability and Security*, F. Koornneef and M. van der Meulen, Eds., ser. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer, 2000, pp. 96–106, ISBN: 978-3-540-40891-8. DOI: 10.1007/3-540-40891-6_9.

[74] G. Oikonomou, S. Duquennoy, A. Elsts, J. Eriksson, Y. Tanaka, and N. Tsiftes, "The contiki-NG open source operating system for next generation IoT devices," *SoftwareX*, vol. 18, Jun. 1, 2022, Publisher: Elsevier, ISSN: 2352-7110. DOI: 10.1016/j.softx.2022.101089. [Online]. Available: https://www.softxjournal.com/article/S2352-7110(22)00062-0/fulltext (visited on 12/26/2022).

[75] PicoTCP. "Picotcp," picotcp. (), [Online]. Available: http://picotcp.altran.be/ (visited on 04/07/2023).

[76] C. Cowan *et al.*, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.," in *USENIX security symposium*, San Antonio, TX, vol. 98, 1998, pp. 63–78.

[77] "Defeating solar designer's non-executable stack patch." (), [Online]. Available: https://insecure.org/sploits/non-executable.stack.problems.html (visited on 08/19/2023).

[78] IEEE, "IEEE standard for IEEE information technology - portable operating system interface (POSIX(TM))," *IEEE Std 1003.1-2001 (Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992)*, pp. 1–3678, Dec. 2001, Conference Name: IEEE Std 1003.1-2001 (Revision of IEEE Std 1003.1-1996 and IEEE Std 1003.2-1992). DOI: 10.1109/IEEESTD.2001.93364.

[79] Robert T. Braden, "Requirements for internet hosts - application and support," Internet Engineering Task Force, Request for Comments RFC 1123, Oct. 1989, Num Pages: 98. DOI: 10.17487/RFC1123. [Online]. Available: https://datatracker.ietf.org/doc/rfc1123 (visited on 02/26/2023).

[80] R. T. Braden, "Requirements for internet hosts - communication layers," Internet Engineering Task Force, Request for Comments RFC 1122, Oct. 1989, Num Pages: 116. DOI: 10.17487/RFC1122. [Online]. Available: https://datatracker.ietf.org/doc/rfc1122 (visited on 08/15/2023).

[81] G. Fraser and A. Arcuri, "EvoSuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11, New York, NY, USA: Association for Computing Machinery, Sep. 9, 2011, pp. 416–419, ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025179. [Online]. Available: https://dl.acm.org/doi/10.1145/2025113.2025179 (visited on 04/12/2023).

[82] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*, ISSN: 1558-1225, May 2007, pp. 75–84. DOI: 10.1109/ICSE.2007.37.

[83] D. Beyer, "Advances in automatic software testing: Test-compă2022," in *International Conference on Fundamental Approaches to Software Engineering*, Berlin, Heidelberg: Springer-Verlag, Apr. 4, 2022, pp. 321–335, ISBN: 978-3-030-99428-0. DOI: 10.1007/978-3-030-99429-7_18. [Online]. Available: https://doi.org/10.1007/978-3-030-99429-7_18 (visited on 08/14/2023).

[84] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H. C. Gall, and A. Bacchelli, "On the effectiveness of manual and automatic unit test generation: Ten years later," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, ISSN: 2574-3864, May 2019, pp. 121–125. DOI: 10.1109/MSR.2019.00028.

[85] N. Cardwell *et al.*, "Packetdrill: Scriptable network stack testing, from sockets to packets," presented at the 2013 USENIX Annual Technical Conference (USENIX ATC 13), 2013, pp. 213–218, ISBN: 978-1-931971-01-0. [Online]. Available: https://www.usenix.org/conference/atc13/technical-sessions/presentation/cardwell (visited on 08/15/2023).

[86] I. InterWorking Labs, *Network testing products*, 2022. [Online]. Available: https://www.iwl.com/products.

[87] N. K. Gopalakrishna, D. Anandayuvaraj, A. Detti, F. L. Bland, S. Rahaman, and J. C. Davis, ""if security is required": Engineering and security practices for machine learning-based IoT devices," in *2022 IEEE/ACM 4th International Workshop on Software Engineering Research and Practices for the IoT (SERP4IoT)*, May 2022, pp. 1–8. DOI: 10.1145/3528227.3528565.

[88] Microsoft. "Project everest," Microsoft Research. (), [Online]. Available: https://www.microsoft.com/en-us/research/project/project-everest-verified-secure-implementations-https-ecosystem/ (visited on 04/07/2023).

[89] N. Chong and B. Jacobs, "Formally verifying freertos' interprocess communication mechanism," in *Embedded World Exhibition & Conference 2021*, 2021. [Online]. Available: https://www.amazon.science/publications/formally-verifying-freertos-interprocess-communication-mechanism.

[90] B. Liu *et al.*, "A large-scale empirical study on vulnerability distribution within projects and the lessons learned," in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, ISSN: 1558-1225, Oct. 2020, pp. 1547–1559.

[91] M. Jimenez, M. Papadakis, and Y. L. Traon, "An empirical analysis of vulnerabilities in OpenSSL and the linux kernel," in *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, ISSN: 1530-1362, Dec. 2016, pp. 105–112. DOI: 10.1109/APSEC.2016.025.

[92] M. Cai, H. Huang, and J. Huang, "Understanding security vulnerabilities in file systems," in *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*, ser. APSys '19, New York, NY, USA: Association for Computing Machinery, Aug. 19, 2019, pp. 8–15, ISBN: 978-1-4503-6893-3. DOI: 10.1145/3343737.3343753. [Online]. Available: https://doi.org/10.1145/3343737.3343753 (visited on 12/27/2021).

[93] A. Al-Boghdady, K. Wassif, and M. El-Ramly, "The presence, trends, and causes of security vulnerabilities in operating systems of IoT's low-end devices," *Sensors*, vol. 21, no. 7, p. 2329, Jan. 2021, Number: 7 Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 1424-8220. DOI: 10.3390/s21072329. [Online]. Available: https://www.mdpi.com/1424-8220/21/7/2329 (visited on 08/18/2023).

[94] J. McBride, B. Arief, and J. Hernandez-Castro, "Security analysis of contiki IoT operating system," in *Proceedings of the 2018 International Conference on Embedded Wireless Systems and Networks*, ser. EWSN ’18, USA: Junction Publishing, Feb. 12, 2018, pp. 278–283, ISBN: 978-0-9949886-2-1. (visited on 08/18/2023).

[95] J. Malik and F. Pastore, "An empirical study of vulnerabilities in edge frameworks to support security testing improvement," *Empirical Software Engineering*, vol. 28, no. 4, p. 99, 2023.

[96] Zimperium Labs. "FreeRTOS TCP/IP stack vulnerabilities - the details," Zimperium. (), [Online]. Available: https://www.zimperium.com/blog/freertos-tcpip-stack-vulnerabilities-details/ (visited on 04/09/2023).

[97] A. Danial, *Cloc: Count lines of code*, https://github.com/AlDanial/cloc, Accessed: May 3, 2023, 2021.

[98] M. Silva, D. Cerdeira, S. Pinto, and T. Gomes, "Operating systems for internet of things low-end devices: Analysis and benchmarking," *IEEE Internet of Things Journal*, vol. 6, no. 6, pp. 10 375–10 383, Dec. 2019, Conference Name: IEEE Internet of Things Journal, ISSN: 2327-4662. DOI: 10.1109/JIOT.2019.2939008.

[99] *National vulnerability database*, https://nvd.nist.gov/, Accessed: May 3, 2023.

[100] P. C. Amusuo, A. Sharma, S. R. Rao, A. Vincent, and J. C. Davis, "Reflections on software failure analysis," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, New York, NY, USA: Association for Computing Machinery, Nov. 9, 2022, pp. 1615–1620, ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3560879. [Online]. Available: https://dl.acm.org/doi/10.1145/3540250.3560879 (visited on 11/11/2024).

[101] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977, Publisher: [Wiley, International Biometric Society], ISSN: 0006-341X. DOI: 10.2307/2529310. [Online]. Available: https://www.jstor.org/stable/2529310 (visited on 05/06/2023).

[102] *Common weakness enumeration*, https://cwe.mitre.org/, Accessed: May 3, 2023.

[103] Contiki, *An introduction to cooja*, https://github.com/contiki-os/contiki/wiki/An-Introduction-to-Cooja, Accessed: May 3, 2023, 2021.

[104] M. Muench, J. Stijohann, F. Kargl, A. Francillon, and D. Balzarotti, "What you corrupt is not what you crash: Challenges in fuzzing embedded devices.," in *NDSS*, 2018.

[105] "Universal TUN/TAP device driver — the linux kernel documentation." (), [Online]. Available: https://docs.kernel.org/networking/tuntap.html (visited on 05/06/2023).

[106] J. Srinivasan, S. R. Tanksalkar, P. C. Amusuo, J. C. Davis, and A. Machiry, "Towards rehosting embedded applications as linux applications," in *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2023.

[107] M. Shen, J. C. Davis, and A. Machiry, "Towards automated identification of layering violations in embedded applications," in *2023 ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, ACM, 2023.

[108] C. Poncelet, K. Sagonas, and N. Tsiftes, "So many fuzzers, so little time*: Experience from evaluating fuzzers on the contiki-NG network (hay)stack," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '22, New York, NY, USA: Association for Computing Machinery, Jan. 5, 2023, pp. 1–12, ISBN: 978-1-4503-9475-8. DOI: 10.1145/3551349.3556946. [Online]. Available: https://dl.acm.org/doi/10.1145/3551349.3556946 (visited on 05/05/2023).

[109] C. Lyu *et al.*, "Mopt: Optimized mutation scheduling for fuzzers.," in *USENIX Security Symposium*, 2019, pp. 1949–1966.

[110] M. Cho, S. Kim, and T. Kwon, "Intriguer: Field-level constraint solving for hybrid fuzzing," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 515–530.

[111] S. Poeplau and A. Francillon, "Symbolic execution with symcc: Don't interpret, compile!" In *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 181–198.

[112] A. S. Ami, K. Moran, D. Poshyvanyk, and A. Nadkarni, *"false negative – that one is going to kill you": Understanding industry perspectives of static analysis based security testing*, Aug. 1, 2023. arXiv: 2307.16325[cs]. [Online]. Available: http://arxiv.org/abs/2307.16325 (visited on 08/11/2023).

[113] *CodeQL*. [Online]. Available: https://codeql.github.com/ (visited on 05/21/2022).

[114] D. Anandayuvaraj and J. C. Davis, "Reflecting on Recurring Failures in IoT Development," in *Automated Software Engineering: New Ideas and Emerging Results (ASE-NIER)*, 2022.

[115] D. Anandayuvaraj, P. Thulluri, J. Figueroa, H. Shandilya, and J. C. Davis, "Incorporating Failure Knowledge into Design Decisions for IoT Systems: A Controlled Experiment on Novices," in *Software Engineering Research & Practices for the Internet of Things (SERP4IoT)*, 2023.

[116] C. Bodei, S. Chessa, and L. Galletta, "Measuring security in IoT communications," *Theoretical Computer Science*, Selected papers of ICTCS 2016 (The Italian Conference on Theoretical Computer Science (ICTCS), vol. 764, pp. 100–124, Apr. 11, 2019, ISSN: 0304-3975. DOI: 10.1016/j.tcs.2018.12.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397518307205 (visited on 12/09/2021).

[117] W. Toussaint and A. Y. Ding, "Machine learning systems in the IoT: Trustworthiness trade-offs for edge intelligence," in *2020 IEEE Second International Conference on Cognitive Machine Intelligence (CogMI)*, Oct. 2020, pp. 177–184. DOI: 10.1109/CogMI50398.2020.00030.

[118] "More about AFL — AFL 2.53b documentation." (), [Online]. Available: https://afl-1.readthedocs.io/en/latest/about_afl.html (visited on 05/06/2023).

[119] "CRIU." (), [Online]. Available: https://criu.org/Main_Page (visited on 05/06/2023).

[120] N. Stephens *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution," in *Proceedings 2016 Network and Distributed System Security Symposium*, San Diego, CA: Internet Society, 2016, ISBN: 978-1-891562-41-9. DOI: 10.14722/ndss.2016.23368. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2017/09/driller-augmenting-fuzzing-through-selective-symbolic-execution.pdf (visited on 01/27/2022).

[121] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM} : A practical concolic execution engine tailored for hybrid fuzzing," presented at the 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 745–761, ISBN: 978-1-939133-04-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/yun (visited on 01/27/2022).

[122] N. Chong *et al.*, "Code-level model checking in the software development workflow," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ser. ICSE-SEIP '20, New York, NY, USA: Association for Computing Machinery, Sep. 18, 2020, pp. 11–20, ISBN: 978-1-4503-7123-0. DOI: 10.1145/3377813.3381347. [Online]. Available: https://dl.acm.org/doi/10.1145/3377813.3381347 (visited on 10/01/2024).

[123] P. C. Amusuo, O. Cochell, T. L. Lievre, P. V. Patil, A. Machiry, and J. C. Davis, "Do unit proofs work? an empirical study of compositional bounded model checking for memory safety verification," in *2026 IEEE/ACM 48th International Conference on Software Engineering*, Mar. 17, 2025. DOI: 10.48550/arXiv.2503.13762. arXiv: 2503.13762[cs]. [Online]. Available: http://arxiv.org/abs/2503.13762 (visited on 09/30/2025).

[124] "Memory safety," The Chromium Projects. (), [Online]. Available: https://www.chromium.org/Home/chromium-security/memory-safety/ (visited on 03/08/2025).

[125] D. Weston. "Helping our customers through the CrowdStrike outage," The Official Microsoft Blog. (Jul. 20, 2024), [Online]. Available: https://blogs.microsoft.com/blog/2024/07/20/helping-our-customers-through-the-crowdstrike-outage/ (visited on 03/08/2025).

[126] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: Rust as a case study," presented at the Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021), 2021, pp. 597–616, ISBN: 978-1-939133-25-0. [Online]. Available: https://www.usenix.org/conference/soups2021/presentation/fulton (visited on 03/08/2025).

[127] "DARPA guide for formal methods to deliver resilient systems for proposals." (), [Online]. Available: https://defencescienceinstitute.com/wp-content/uploads/2025/01/Resilient_Systems_Best_Practices_Guide_-_1-9-2025.pdf (visited on 03/08/2025).

[128] T. Wu, S. Xiong, E. Manino, G. Stockwell, and L. C. Cordeiro. "Verifying components of arm(r) confidential computing architecture with ESBMC," arXiv.org. (Jun. 5, 2024), [Online]. Available: https://arxiv.org/abs/2406.04375v1 (visited on 10/04/2024).

[129] M. Wang, J. Xue, L. Huang, Y. Zi, and T. Wei, "UnsafeCop: Towards memory safety forărеal-world unsafe rust code withăpractical bounded model checking," in *Formal Methods*, A. Platzer, K. Y. Rozier, M. Pradella, and M. Rossi, Eds., Cham: Springer Nature Switzerland, 2025, pp. 307–324, ISBN: 978-3-031-71177-0. DOI: 10.1007/978-3-031-71177-0_19.

[130] P. C. Amusuo, P. V. Patil, O. Cochell, T. L. Lievre, and J. C. Davis, *Enabling unit proofing for software implementation verification*, Oct. 18, 2024. DOI: 10.48550/arXiv.2410.14818. arXiv: 2410.14818. [Online]. Available: http://arxiv.org/abs/2410.14818 (visited on 11/11/2024).

[131] D. Matichuk, T. Murray, J. Andronick, R. Jeffery, G. Klein, and M. Staples, "Empirical study towards a leading indicator for cost of formal software verification," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, ISSN: 1558-1225, vol. 1, May 2015, pp. 722–732. DOI: 10.1109/ICSE.2015.85. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/7194620 (visited on 09/21/2024).

[132] M. Olszewska and K. Sere, "Specification metrics for event-b developments," presented at the CONQUEST 2010, Dresden, Germany, Sep. 20, 2010, pp. 1–12. [Online]. Available: https://web-archive.southampton.ac.uk/deploy-eprints.ecs.soton.ac.uk/249/ (visited on 02/18/2025).

[133] D. Beyer and T. Lemberger, "Software verification: Testing vs. model checking," in *Hardware and Software: Verification and Testing*, O. Strichman and R. Tzoref-Brill, Eds., Cham: Springer International Publishing, 2017, pp. 99–114, ISBN: 978-3-319-70389-3. DOI: 10.1007/978-3-319-70389-3_7.

[134] 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*, vol. 46, no. 2, pp. 265–288, 2007, Conference Name: IBM Systems Journal, ISSN: 0018-8670. DOI: 10.1147/sj.462.0265. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5386616 (visited on 10/18/2024).

[135] T. Klooster, F. Turkmen, G. Broenink, R. T. Hove, and M. Böhme, "Continuous fuzzing: A study of the effectiveness and scalability of fuzzing in CI/CD pipelines," in *2023 IEEE/ACM International Workshop on Search-Based and Fuzz Testing (SBFT)*, May 2023, pp. 25–32. DOI: 10.1109/SBFT59156.2023.00015. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10190386 (visited on 11/11/2024).

[136] L. Davi, A.-R. Sadeghi, and M. Winandy, "ROPdefender: A detection tool to defend against return-oriented programming attacks," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '11, New York, NY, USA: Association for Computing Machinery, Mar. 22, 2011, pp. 40–51, ISBN: 978-1-4503-0564-8. DOI: 10.1145/1966913.1966920. [Online]. Available: https://dl.acm.org/doi/10.1145/1966913.1966920 (visited on 03/10/2025).

[137] C. Cowan, S. Beattie, R. F. Day, C. Pu, P. Wagle, and E. Walthinsen, "Protecting systems from stack smashing attacks with StackGuard,"

[138] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "{ACES}: Automatic compartments for embedded systems," presented at the 27th USENIX Security Symposium (USENIX Security 18), 2018, pp. 65–82, ISBN: 978-1-939133-04-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/clements (visited on 08/08/2023).

[139] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded model checking using satisfiability solving," *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, Jul. 1, 2001, ISSN: 1572-8102. DOI: 10.1023/A:1011276507260. [Online]. Available: https://doi.org/10.1023/A:1011276507260 (visited on 10/03/2024).

[140] Martin Brain. "CBMC: Goto-cc." (), [Online]. Available: https://diffblue.github.io/cbmc/group___goto-cc.html (visited on 09/26/2024).

[141] D. Kroening and M. Tautschnig, "CBMC – c bounded model checker," in *Tools and Algorithms for the Construction and Analysis of Systems*, E. Ábrahám and K. Havelund, Eds., Berlin, Heidelberg: Springer, 2014, pp. 389–391, ISBN: 978-3-642-54862-8. DOI: 10.1007/978-3-642-54862-8_26.

[142] kani. "Getting started - the kani rust verifier." (), [Online]. Available: https://model-checking.github.io/kani/ (visited on 10/01/2024).

[143] "Reference manual - CBMC viewer." (), [Online]. Available: https://model-checking.github.io/cbmc-viewer/reference-manual/ (visited on 03/08/2025).

[144] *Model-checking/cbmc-proof-debugger*, original-date: 2022-10-29T15:25:12Z, Oct. 27, 2024. [Online]. Available: https://github.com/model-checking/cbmc-proof-debugger (visited on 03/08/2025).

[145] "The CPROVER manual - goto harness." (), [Online]. Available: https://www.cprover.org/cprover-manual/goto-harness/ (visited on 03/08/2025).

[146] T. J. Schaefer, "The complexity of satisfiability problems," in *Proceedings of the tenth annual ACM symposium on Theory of computing*, ser. STOC '78, New York, NY, USA: Association for Computing Machinery, May 1, 1978, pp. 216–226, ISBN: 978-1-4503-7437-8. DOI: 10.1145/800133.804350. [Online]. Available: https://dl.acm.org/doi/10.1145/800133.804350 (visited on 03/08/2025).

[147] J. Sun, Y. Liu, J. S. Dong, and J. Sun, "Compositional encoding for bounded model checking," *Frontiers of Computer Science in China*, vol. 2, no. 4, pp. 368–379, Dec. 1, 2008, ISSN: 1673-7466. DOI: 10.1007/s11704-008-0035-6. [Online]. Available: https://doi.org/10.1007/s11704-008-0035-6 (visited on 03/08/2025).

[148] M. Kleine Büning, J. Meuer, and C. Sinz, "Refined modularization forăbounded model checking through precondition generation," in *Formal Methods and Software Engineering*, A. Riesco and M. Zhang, Eds., Cham: Springer International Publishing, 2022, pp. 209–226, ISBN: 978-3-031-17244-1. DOI: 10.1007/978-3-031-17244-1_13.

[149] C. Y. Cho, V. D'Silva, and D. Song, "BLITZ: Compositional bounded model checking for real-world programs," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2013, pp. 136–146. DOI: 10.1109/ASE.2013.6693074. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6693074 (visited on 09/26/2024).

[150] B. Beckert, M. Kirsten, J. Klamroth, and M. Ulbrich, "Modular verification of JML contracts using bounded model checking," in *Leveraging Applications of Formal Methods, Verification and Validation: Verification Principles*, T. Margaria and B. Steffen, Eds., Cham: Springer International Publishing, 2020, pp. 60–80, ISBN: 978-3-030-61362-4. DOI: 10.1007/978-3-030-61362-4_4.

[151] M. Kleine Büning and C. Sinz, "Automatic modularization of large programs for bounded model checking," in *Formal Methods and Software Engineering*, Y. Ait-Ameur and S. Qin, Eds., Cham: Springer International Publishing, 2019, pp. 186–202, ISBN: 978-3-030-32409-4. DOI: 10.1007/978-3-030-32409-4_12.

[152] Y. Hashimoto and S. Nakajima, "Modular checking of c programs using SAT-based bounded model checker," in *2009 16th Asia-Pacific Software Engineering Conference*, ISSN: 1530-1362, Dec. 2009, pp. 515–522. DOI: 10.1109/APSEC.2009.24. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/5358870 (visited on 02/23/2025).

[153] L. Huang, B. Meyer, and R. Weber, *Loop unrolling: Formal definition and application to testing*, Feb. 21, 2025. DOI: 10.48550/arXiv.2502.15535. arXiv: 2502.15535[cs]. [Online]. Available: http://arxiv.org/abs/2502.15535 (visited on 03/09/2025).

[154] J. J. Tsai and K. Xu, "A comparative study of formal verification techniques for software architecture specifications," *Annals of Software Engineering*, vol. 10, no. 1, pp. 207–223, Nov. 1, 2000, ISSN: 1573-7489. DOI: 10.1023/A:1018960305057. [Online]. Available: https://doi.org/10.1023/A:1018960305057 (visited on 02/18/2025).

[155] "RIOT - the friendly operating system for the internet of things." (), [Online]. Available: https://www.riot-os.org/ (visited on 03/10/2025).

[156] "OSRTOS - top open source real-time operating systems (RTOS)." (), [Online]. Available: https://osrtos.com/ (visited on 03/10/2025).

[157] A. Das, S. K. Lahiri, A. Lal, and Y. Li, "Angelic verification: Precise verification modulo unknowns," in *Computer Aided Verification*, D. Kroening and C. S. Păsăreanu, Eds., Cham: Springer International Publishing, 2015, pp. 324–342, ISBN: 978-3-319-21690-4. DOI: 10.1007/978-3-319-21690-4_19.

[158] J. M. Cobleigh, G. S. Avrunin, and L. A. Clarke, "Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning," *ACM Trans. Softw. Eng. Methodol.*, vol. 17, no. 2, 7:1–7:52, May 5, 2008, ISSN: 1049-331X. DOI: 10.1145/1348250.1348253. [Online]. Available: https://dl.acm.org/doi/10.1145/1348250.1348253 (visited on 10/06/2024).

[159] J. M. Cobleigh, D. Giannakopoulou, and C. S. PĂsĂreanu, "Learning assumptions for compositional verification," in *Tools and Algorithms for the Construction and Analysis of Systems*, H. Garavel and J. Hatcliff, Eds., Berlin, Heidelberg: Springer, 2003, pp. 331–346, ISBN: 978-3-540-36577-8. DOI: 10.1007/3-540-36577-X_24.

[160] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang, "Compositional shape analysis by means of bi-abduction," in *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '09, New York, NY, USA: Association for Computing Machinery, Jan. 21, 2009, pp. 289–300, ISBN: 978-1-60558-379-2. DOI: 10.1145/1480881.1480917. [Online]. Available: https://dl.acm.org/doi/10.1145/1480881.1480917 (visited on 03/11/2025).

[161] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, Dec. 1, 1997, ISSN: 0360-0300. DOI: 10.1145/267580.267590. [Online]. Available: https://dl.acm.org/doi/10.1145/267580.267590 (visited on 03/11/2025).

[162] J. Horgan, S. London, and M. Lyu, "Achieving software quality with testing coverage measures," *Computer*, vol. 27, no. 9, pp. 60–69, Sep. 1994, Conference Name: Computer, ISSN: 1558-0814. DOI: 10.1109/2.312032. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/312032 (visited on 03/11/2025).

[163] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," presented at the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019), 2019, pp. 1–15, ISBN: 978-1-939133-07-6. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/wang (visited on 03/11/2025).

[164] M. Böhme, L. Szekeres, and J. Metzman, "On the reliability of coverage-based fuzzer benchmarking," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, New York, NY, USA: Association for Computing Machinery, Jul. 5, 2022, pp. 1621–1633, ISBN: 978-1-4503-9221-1. DOI: 10.1145/3510003.3510230. [Online]. Available: https://dl.acm.org/doi/10.1145/3510003.3510230 (visited on 03/11/2025).

[165] "CBMC starter kit: Proof-writing guide," GitHub. (), [Online]. Available: https://github.com/model-checking/cbmc-starter-kit/blob/master/training-material/PROOF-WRITING.md (visited on 03/11/2025).

[166] *FreeRTOS/FreeRTOS*, original-date: 2019-09-03T16:25:27Z, Mar. 10, 2025. [Online]. Available: https://github.com/FreeRTOS/FreeRTOS (visited on 03/10/2025).

[167] T. Do, T. Harter, Y. Liu, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "{HARDFS}: Hardening {HDFS} with selective and lightweight versioning," presented at the 11th USENIX Conference on File and Storage Technologies (FAST 13), 2013, pp. 105–118, ISBN: 978-1-931971-99-7. [Online]. Available: https://www.usenix.org/conference/fast13/technical-sessions/presentation/do (visited on 03/11/2025).

[168] J.-C. Lin and K.-C. Wu, "Evaluation of software understandability based on fuzzy matrix," in *2008 IEEE International Conference on Fuzzy Systems (IEEE World Congress on Computational Intelligence)*, ISSN: 1098-7584, Jun. 2008, pp. 887–892. DOI: 10.1109/FUZZY.2008.4630475. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4630475 (visited on 03/11/2025).

[169] D. A. Boehm-Davis, "Chapter 5 - software comprehension," in *Handbook of Human-Computer Interaction*, M. Helander, Ed., Amsterdam: North-Holland, Jan. 1, 1988, pp. 107–121, ISBN: 978-0-444-70536-5. DOI: 10.1016/B978-0-444-70536-5.50010-5. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780444705365500105 (visited on 03/11/2025).

[170] V. Antinyan, J. Derehag, A. Sandberg, and M. Staron, "Mythical unit test coverage," *IEEE Software*, vol. 35, no. 3, pp. 73–79, May 2018, Conference Name: IEEE Software, ISSN: 1937-4194. DOI: 10.1109/MS.2017.3281318. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8354427 (visited on 03/11/2025).

[171] T. Yin, *Terryyin/lizard*, original-date: 2012-06-21T11:31:46Z, Mar. 11, 2025. [Online]. Available: https://github.com/terryyin/lizard (visited on 03/12/2025).

[172] "West (zephyr's meta-tool) — zephyr project documentation." (), [Online]. Available: https://docs.zephyrproject.org/latest/develop/west/index.html (visited on 03/13/2025).

[173] P. W. O'Hearn, "Continuous reasoning: Scaling the impact of formal methods," in *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, ser. LICS '18, New York, NY, USA: Association for Computing Machinery, Jul. 9, 2018, pp. 13–25, ISBN: 978-1-4503-5583-4. DOI: 10.1145/3209108.3209109. [Online]. Available: https://dl.acm.org/doi/10.1145/3209108.3209109 (visited on 03/13/2025).

[174] "RIOT/sys/net/gnrc/routing/rpl/gnrc_rpl_control_messages.c at master · RIOT-OS/RIOT." (2025), [Online]. Available: https://github.com/RIOT-OS/RIOT/blob/1407d4b/sys/net/gnrc/routing/rpl/gnrc_rpl_control_messages.c#L515.

[175] A. Farzan, P. Madhusudan, N. Razavi, and F. Sorrentino, "Predicting null-pointer dereferences in concurrent programs," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12, New York, NY, USA: Association for Computing Machinery, Nov. 11, 2012, pp. 1–11, ISBN: 978-1-4503-1614-9. DOI: 10.1145/2393596.2393651. [Online]. Available: https://dl.acm.org/doi/10.1145/2393596.2393651 (visited on 03/13/2025).

[176] R. Xu *et al.*, "Accelerating static null pointer dereference detection with parallel computing," in *Proceedings of the 15th Asia-Pacific Symposium on Internetware*, ser. Internetware '24, New York, NY, USA: Association for Computing Machinery, Jul. 24, 2024, pp. 135–144, ISBN: 9798400707056. DOI: 10.1145/3671016.3671385. [Online]. Available: https://dl.acm.org/doi/10.1145/3671016.3671385 (visited on 03/13/2025).

[177] S. Padhi, R. Sharma, and T. Millstein, "Data-driven precondition inference with learned features," *SIGPLAN Not.*, vol. 51, no. 6, pp. 42–56, Jun. 2, 2016, ISSN: 0362-1340. DOI: 10.1145/2980983.2908099. [Online]. Available: https://dl.acm.org/doi/10.1145/2980983.2908099 (visited on 03/13/2025).

[178] D. Brumley, H. Wang, S. Jha, and D. Song, "Creating vulnerability signatures using weakest preconditions," in *20th IEEE Computer Security Foundations Symposium (CSF'07)*, ISSN: 2377-5459, Jul. 2007, pp. 311–325. DOI: 10.1109/CSF.2007.17. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/4271657 (visited on 03/13/2025).

[179] M. N. Seghir and D. Kroening, "Counterexample-guided precondition inference," in *Programming Languages and Systems*, M. Felleisen and P. Gardner, Eds., Berlin, Heidelberg: Springer, 2013, pp. 451–471, ISBN: 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_25.

[180] Alex Rebert, Chandler Carruth, Jen Engel, and Andy Qin, *Safer with Google: Advancing Memory Safety*, en, 2024. [Online]. Available: https://security.googleblog.com/2024/10/safer-with-google-advancing-memory.html.

[181] MSRC Team, *A proactive approach to more secure code MSRC Blog Microsoft Security Response Center*, 2019. [Online]. Available: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/.

[182] S. Ritvik Tanksalkar *et al.*, "Lemix: Enabling testing of embedded applications as linux applications," in *[USENIX Security'25] USENIX Security Symposium, 2025*, 2025, arXiv–2503.

[183] CrowdStrike, Inc., "External technical root cause analysis — channel file 291," CrowdStrike, Tech. Rep., Aug. 2024, Accessed January 2026. [Online]. Available: https://www.crowdstrike.com/wp-content/uploads/2024/08/Channel-File-291-Incident-Root-Cause-Analysis-08.06.2024.pdf.

[184] Latham & Watkins LLP, *New EU product liability directive comes into force*, Client alert / legal briefing, Accessed January 2026, Dec. 2024. [Online]. Available: https://www.lw.com/admin/upload/SiteAttachments/New-EU-Product-Liability-Directive-Comes-Into-Force.pdf.

[185] J.-C. Filliâtre, "Deductive software verification," *International Journal on Software Tools for Technology Transfer*, vol. 13, no. 5, pp. 397–403, Oct. 1, 2011, ISSN: 1433-2787. DOI: 10.1007/s10009-011-0211-0. [Online]. Available: https://doi.org/10.1007/s10009-011-0211-0 (visited on 04/29/2026).

[186] S. Li, L. Qiao, and M. Yang, "Memory state verification based on inductive and deductive reasoning," *IEEE Transactions on Reliability*, vol. 70, no. 3, pp. 1026–1039, Sep. 2021, ISSN: 1558-1721. DOI: 10.1109/TR.2021.3074709. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9435092 (visited on 04/29/2026).

[187] "IEC 61508-3: Functional safety of electrical/electronic/programmable electronic safety-related systems – part 3: Software requirements." (2010), [Online]. Available: https://webstore.iec.ch/en/publication/5517 (visited on 07/24/2026).

[188] P. C. Amusuo, P. V. Patil, O. Cochell, T. Le Lievre, and J. C. Davis, "A unit proofing framework for code-level verification: A research agenda," in *2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, Apr. 2025, pp. 36–40. DOI: 10.1109/ICSE-NIER66352.2025.00013. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11023946 (visited on 09/30/2025).

[189] Y. Ji, T. Dai, Z. Zhou, Y. Tang, and J. He, "Artemis: Toward accurate detection of server-side request forgeries through LLM-assisted inter-procedural path-sensitive taint analysis," *Artemis: Toward Accurate Detection of Server-Side Request Forgeries through LLM-Assisted Inter-Procedural Path-Sensitive Taint Analysis (Artifact)*, vol. 9, 128:1349–128:1377, OOPSLA1 Apr. 9, 2025. DOI: 10.1145/3720488. [Online]. Available: https://dl.acm.org/doi/10.1145/3720488 (visited on 04/30/2026).

[190] Z. Li, S. Dutta, and M. Naik, "IRIS: LLM-assisted static analysis for detecting security vulnerabilities," in *International Conference on Learning Representations*, Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu, Eds., vol. 2025, 2025, pp. 35 735–35 758. [Online]. Available: https://proceedings.iclr.cc/paper_files/paper/2025/file/582d4e27fa24168f3af1f4582655034b-Paper-Conference.pdf.

[191] A. Azevedo de Amorim, C. Hriţcu, and B. C. Pierce, "The meaning of memory safety," in *Principles of Security and Trust*, L. Bauer and R. Küsters, Eds., Cham: Springer International Publishing, 2018, pp. 79–105, ISBN: 978-3-319-89722-6. DOI: 10.1007/978-3-319-89722-6_4.

[192] M. Hicks. "What is memory safety? - the PL enthusiast," The Programming Languages Enthusiast. (Jul. 21, 2014), [Online]. Available: http://www.pl-enthusiast.net/2014/07/21/memory-safety/ (visited on 04/06/2026).

[193] P. C. van Oorschot, "Memory errors and memory safety: C as a case study," *IEEE Security & Privacy*, vol. 21, no. 2, pp. 70–76, Mar. 2023, ISSN: 1558-4046. DOI: 10.1109/MSEC.2023.3236542. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10102611 (visited on 04/06/2026).

[194] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, and P. Ashar, "Efficient SAT-based bounded model checking for software verification," *Theoretical Computer Science*, International Symposium on Leveraging Applications of Formal Methods (ISoLA 2004), vol. 404, no. 3, pp. 256–274, Sep. 28, 2008, ISSN: 0304-3975. DOI: 10.1016/j.tcs.2008.03.013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0304397508002223 (visited on 02/04/2026).

[195] A. Mashkoor, M. Leuschel, and A. Egyed, "Validation obligations: A novel approach to check compliance between requirements and their formal specification," in *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, May 2021, pp. 1–5. DOI: 10.1109/ICSE-NIER52604.2021.00009. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9402243 (visited on 04/29/2026).

[196] B. Feng, A. Mera, and L. Lu, "{P2im}: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," presented at the 29th USENIX Security Symposium (USENIX Security 20), 2020, pp. 1237–1254, ISBN: 978-1-939133-17-5. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/feng (visited on 11/02/2023).

[197] F. Ivančić *et al.*, "DC2: A framework for scalable, scope-bounded software verification," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, ISSN: 1938-4300, Nov. 2011, pp. 133–142. DOI: 10.1109/ASE.2011.6100046. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6100046 (visited on 02/03/2026).

[198] G. Takhar, B. Bijlani, P. Chatterjee, A. Lal, and S. Roy, "Memory-safety verification of open programs with angelic assumptions," *Memory-Safety Verification of Open Programs With Angelic Assumptions*, vol. 9, 312:1119–312:1147, OOPSLA2 Oct. 9, 2025. DOI: 10.1145/3763090. [Online]. Available: https://dl.acm.org/doi/10.1145/3763090 (visited on 02/03/2026).

[199] M. Y. R. Gadelha, H. I. Ismail, and L. C. Cordeiro, "Handling loops in bounded model checking of c programs via k-induction," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 1, pp. 97–114, Feb. 1, 2017, ISSN: 1433-2787. DOI: 10.1007/s10009-015-0407-9. [Online]. Available: https://doi.org/10.1007/s10009-015-0407-9 (visited on 04/29/2026).

[200] *Openai/openai-python*, original-date: 2020-10-25T23:23:54Z, Feb. 6, 2026. [Online]. Available: https://github.com/openai/openai-python (visited on 02/06/2026).

[201] "LiteLLM - getting started liteLLM." (), [Online]. Available: https://docs.litellm.ai/docs/ (visited on 02/06/2026).

[202] "SWE-bench leaderboards." (), [Online]. Available: https://www.swebench.com/index.html (visited on 04/30/2026).

[203] B. Robinson, M. D. Ernst, J. H. Perkins, V. Augustine, and N. Li, "Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, ISSN: 1938-4300, Nov. 2011, pp. 23–32. DOI: 10.1109/ASE.2011.6100059. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6100059 (visited on 04/30/2026).

[204] E. Daka, J. Campos, G. Fraser, J. Dorn, and W. Weimer, "Modeling readability to improve unit tests," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, New York, NY, USA: Association for Computing Machinery, Aug. 30, 2015, pp. 107–118, ISBN: 978-1-4503-3675-8. DOI: 10.1145/2786805.2786838. [Online]. Available: https://dl.acm.org/doi/10.1145/2786805.2786838 (visited on 04/29/2026).

[205] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of c programs," in *NASA Formal Methods*, M. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, Eds., Berlin, Heidelberg: Springer, 2011, pp. 459–465, ISBN: 978-3-642-20398-5. DOI: 10.1007/978-3-642-20398-5_33.

[206] M. Pitchford, "The 'shift left' principle," *New Electronics*, vol. 54, no. 14, pp. 18–21, Sep. 2021, Publisher: Mark Allen Group, ISSN: 0047-9624. DOI: 10.12968/S0047-9624(22)60234-7. [Online]. Available: https://www.magonlinelibrary.com/doi/full/10.12968/S0047-9624%2822%2960234-7 (visited on 11/11/2024).

[207] "Software defect reduction top 10 list." (), [Online]. Available: https://www.computer.org/csdl/magazine/co/2001/01/r1135/13rRUwgyOg9 (visited on 04/30/2026).

[208] H. Lefeuvre, V.-A. Bădoiu, Y. Chen, F. Huici, N. Dautenhahn, and P. Olivier, "Assessing the impact of interface vulnerabilities in compartmentalized software," in *Proceedings 2023 Network and Distributed System Security Symposium*, San Diego, CA, USA: Internet Society, 2023, ISBN: 978-1-891562-83-9. DOI: 10.14722/ndss.2023.24117. (visited on 04/30/2026).

[209] M. A. A. Pirzada, G. Reger, A. Bhayat, and L. C. Cordeiro, "LLM-generated invariants for bounded model checking without loop unrolling," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24, New York, NY, USA: Association for Computing Machinery, Oct. 27, 2024, pp. 1395–1407, ISBN: 979-8-4007-1248-7. DOI: 10.1145/3691620.3695512. [Online]. Available: https://dl.acm.org/doi/10.1145/3691620.3695512 (visited on 04/29/2026).

[210] S. Bensalem, M. Bozga, T.-H. Nguyen, and J. Sifakis, "Compositional verification for component-based systems and application," *IET Software*, vol. 4, no. 3, pp. 181–193, Jun. 1, 2010, Publisher: IET Digital Library, ISSN: 1751-8814. DOI: 10.1049/iet-sen.2009.0011. [Online]. Available: https://digital-library.theiet.org/content/journals/10.1049/iet-sen.2009.0011 (visited on 10/01/2024).

[211] T. Le-Cong, B. Le, and T. Murray, "Can LLMs reason about program semantics? a comprehensive evaluation of LLMs on formal specification inference," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds., Vienna, Austria: Association for Computational Linguistics, Jul. 2025, pp. 21 991–22 014, ISBN: 979-8-89176-251-0. DOI: 10.18653/v1/2025.acl-long.1068. [Online]. Available: https://aclanthology.org/2025.acl-long.1068/ (visited on 04/30/2026).

[212] S. R. Kasibatla, A. Agarwal, Y. Brun, S. Lerner, T. Ringer, and E. First. "Cobblestone: A divide-and-conquer approach for automating formal verification," arXiv.org. (Oct. 25, 2024), [Online]. Available: https://arxiv.org/abs/2410.19940v4 (visited on 04/30/2026).

[213] C. Yang *et al.* "AutoVerus: Automated proof generation for rust code," arXiv.org. (Sep. 19, 2024), [Online]. Available: https://arxiv.org/abs/2409.13082v1 (visited on 09/26/2024).

[214] E. First, M. N. Rabe, T. Ringer, and Y. Brun, "Baldur: Whole-proof generation and repair with large language models," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023, New York, NY, USA: Association for Computing Machinery, Nov. 30, 2023, pp. 1229–1241, ISBN: 9798400703270. DOI: 10.1145/3611643.3616243. [Online]. Available: https://dl.acm.org/doi/10.1145/3611643.3616243 (visited on 10/07/2024).

[215] N. Tihanyi, Y. Charalambous, R. Jain, M. A. Ferrag, and L. C. Cordeiro, "A new era in software security: Towards self-healing software via large language models and formal verification," in *2025 IEEE/ACM International Conference on Automation of Software Test (AST)*, Ottawa, ON, Canada: IEEE Press, Apr. 28, 2025, pp. 136–147. DOI: 10.1109/AST66626.2025.00020. [Online]. Available: https://dl.acm.org/doi/10.1109/AST66626.2025.00020 (visited on 04/29/2026).

[216] C. Zhang *et al.*, "How effective are they? exploring large language model based fuzz driver generation," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024, New York, NY, USA: Association for Computing Machinery, Sep. 11, 2024, pp. 1223–1235, ISBN: 9798400706127. DOI: 10.1145/3650212.3680355. [Online]. Available: https://dl.acm.org/doi/10.1145/3650212.3680355 (visited on 01/08/2025).

[217] P. C. Amusuo, D. Liu, R. A. C. Mendez, J. Metzman, O. Chang, and J. C. Davis, *FalseCrashReducer: Mitigating false positive crashes in OSS-fuzz-gen using agentic AI*, Oct. 2, 2025. DOI: 10.48550/arXiv.2510.02185. arXiv: 2510.02185[cs]. [Online]. Available: http://arxiv.org/abs/2510.02185 (visited on 04/30/2026).

[218] H. Xu *et al.*, "CKGFuzzer: LLM-based fuzz driver generation enhanced by code knowledge graph," in *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, Apr. 2025, pp. 243–254. DOI: 10.1109/ICSE-Companion66252.2025.00079. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11024256 (visited on 09/25/2025).

[219] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at facebook," *Communications of the ACM*, vol. 62, no. 8, pp. 62–70, Jul. 24, 2019, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3338112. [Online]. Available: https://dl.acm.org/doi/10.1145/3338112 (visited on 11/11/2024).

[220] C. Zhang, Y. Wang, and L. Wang, "Firmware fuzzing: The state of the art," in *12th Asia-Pacific Symposium on Internetware*, ser. Internetware'20, New York, NY, USA: Association for Computing Machinery, Nov. 1, 2020, pp. 110–115, ISBN: 978-1-4503-8819-1. DOI: 10.1145/3457913.3457934. [Online]. Available: https://doi.org/10.1145/3457913.3457934 (visited on 11/30/2021).

[221] D. Maier, L. Seidel, and S. Park, "BaseSAFE: Baseband sanitized fuzzing through emulation," in *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, ser. WiSec '20, New York, NY, USA: Association for Computing Machinery, Jul. 21, 2020, pp. 122–132, ISBN: 978-1-4503-8006-5. DOI: 10.1145/3395351.3399360. [Online]. Available: https://dl.acm.org/doi/10.1145/3395351.3399360 (visited on 02/05/2026).

[222] Y. Zheng, Y. Li, C. Zhang, H. Zhu, Y. Liu, and L. Sun, "Efficient greybox fuzzing of applications in linux-based IoT devices via enhanced user-mode emulation," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022, New York, NY, USA: Association for Computing Machinery, Jul. 18, 2022, pp. 417–428, ISBN: 978-1-4503-9379-9. DOI: 10.1145/3533767.3534414. [Online]. Available: https://dl.acm.org/doi/10.1145/3533767.3534414 (visited on 02/05/2026).

[223] M. Bley, T. Scharnowski, S. Wörner, M. Schloegel, and T. Holz, "Protocol-aware firmware rehosting for effective fuzzing of embedded network stacks," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '25, New York, NY, USA: Association for Computing Machinery, Nov. 22, 2025, pp. 4484–4498, ISBN: 979-8-4007-1525-9. DOI: 10.1145/3719027.3765125. [Online]. Available: https://dl.acm.org/doi/10.1145/3719027.3765125 (visited on 02/05/2026).

[224] A. Fasano *et al.*, "SoK: Enabling security analyses of embedded systems via rehosting," ser. ASIA CCS '21, New York, NY, USA: Association for Computing Machinery, Jun. 4, 2021, pp. 687–701, ISBN: 978-1-4503-8287-8. DOI: 10.1145/3433210.3453093. [Online]. Available: https://dl.acm.org/doi/10.1145/3433210.3453093 (visited on 10/24/2023).

[225] M. R. H. Misu, C. V. Lopes, I. Ma, and J. Noble, "Towards AI-assisted synthesis of verified dafny methods," *Artifacts@FSE24: Towards AI-Assisted Synthesis of Verified Dafny Methods*, vol. 1, 37:812–37:835, FSE Jul. 12, 2024. DOI: 10.1145/3643763. [Online]. Available: https://dl.acm.org/doi/10.1145/3643763 (visited on 10/07/2024).

[226] M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, ser. NSDI'04, USA: USENIX Association, Mar. 29, 2004, p. 12. (visited on 07/10/2026).

[227] M. Musuvathi, S. Qadeer, P. A. Nainar, T. Ball, G. Basler, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs," in *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008*, USENIX Association, Jan. 1, 2019, pp. 267–280. [Online]. Available: https://research with.njit.edu/en/publications/finding-and-reproducing-heisenbugs-in-concurrent-programs/ (visited on 07/10/2026).

[228] C. Flanagan and S. Qadeer, "Thread-modular model checking," in *Model Checking Software*, T. Ball and S. K. Rajamani, Eds., Berlin, Heidelberg: Springer, 2003, pp. 213–224, ISBN: 978-3-540-44829-7. DOI: 10.1007/3-540-44829-2_14.

[229] "Claude fable 5 and claude mythos 5." (), [Online]. Available: https://www.anthropic.com/news/claude-fable-5-mythos-5 (visited on 07/24/2026).

[230] "GPT-5.6: Frontier intelligence that scales with your ambition," OpenAI. (Jul. 23, 2026), [Online]. Available: https://openai.com/index/gpt-5-6/ (visited on 07/24/2026).

[231] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An empirical study of the non-determinism of ChatGPT in code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, 42:1–42:28, Jan. 22, 2025, ISSN: 1049-331X. DOI: 10.1145/3697010. [Online]. Available: https://dl.acm.org/doi/10.1145/3697010 (visited on 07/24/2026).

[232] N. Mündler, J. He, H. Wang, K. Sen, D. Song, and M. Vechev, "Type-constrained code generation with language models," *Proceedings of the ACM on Programming Languages*, vol. 9, 171:601–171:626, PLDI Jun. 13, 2025. DOI: 10.1145/3729274. [Online]. Available: https://dl.acm.org/doi/10.1145/3729274 (visited on 07/24/2026).

[233] Y. Zhu *et al.*, "Teams of LLM agents can exploit zero-day vulnerabilities," in *Proceedings of the 19th Conference of the European Chapter of the Association for Computational Linguistics (Volume 1: Long Papers)*, V. Demberg, K. Inui, and L. Marquez, Eds., Rabat, Morocco: Association for Computational Linguistics, Mar. 2026, pp. 23–35, ISBN: 979-8-89176-380-7. DOI: 10.18653/v1/2026.eacl-long.2. [Online]. Available: https://aclanthology.org/2026.eacl-long.2/ (visited on 07/24/2026).

[234] "Assessing claude mythos preview's cybersecurity capabilities." (), [Online]. Available: https://www.anthropic.com/research/mythos-preview (visited on 07/24/2026).