On the Path to Buffer Overflow Detection by Model Checking the Stack of Binary Programs

Luís Ferreirinha[a] and Ibéria Medeiros[b]
LASIGE, Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa, Portugal
lpferreirinha@fc.ul.pt, imedeiros@di.fc.ul.pt

Keywords: Stack Buffer Overflow, Assembly, Model Checking, Linear Temporal Logic, Static Analysis, Software Security

Abstract: The C programming language, prevalent in Cyber-Physical Systems, is crucial for system control where reliability is critical. However, it is notably susceptible to vulnerabilities, particularly buffer overflows that are ranked among the most dangerous due to their potential for catastrophic consequences. Traditional techniques, such as static analysis, often struggle with scalability and precision when detecting these vulnerabilities in the binary code of compiled C programs. This paper introduces a novel approach designed to overcome these limitations by leveraging model checking techniques to verify security properties within a program's stack memory. To verify these properties, we propose the construction of a *state space of the stack memory* from a binary program's control flow graph. Security properties, modelled for stack buffer overflow vulnerabilities and defined in Linear Temporal Logic, are verified against this state space. When violations are detected, counter-example traces are generated to undergo a reverse-flow analysis process to identify specific instances of stack buffer overflow vulnerabilities. This research aims to provide a scalable and precise approach to vulnerability detection in C binaries.

1 INTRODUCTION

Software powers the systems of our world, from the smallest gadgets to the largest machines in our industries. It is essential that this software does not just work, but works without fail, preventing errors that could lead to serious consequences. As our reliance on technology grows, so does the need for software that is not just functional, but secure and dependable. Most of this software is written in the C programming language, particularly in cyber-physical systems where reliability is crucial. C allows programmers to work close to the system's hardware, allowing for greater flexibility, but this comes with significant risks. The language leaves room for vulnerabilities such as buffer overflows (BO), where the lack of safeguards can lead to system compromises and failures. These vulnerabilities occur when a write operation is performed outside the bounds of a buffer, and are especially dangerous. As shown by Aleph One (One, 1996), through a BO, an attacker can hijack the flow of execution of the program and execute arbitrary code, allowing full access to the system.

Some efforts (Inácio and Medeiros, 2023; Kroes et al., 2018) have been made to develop methods for detecting such vulnerabilities, with most employing static or dynamic analysis techniques, and in some cases, a hybrid of both. Static analysis examines a program's code without executing it and achieves higher code coverage but at the cost of a higher number of false positives (Nadeem et al., 2012). On the other hand, dynamic analysis executes the program's code, offering more accurate vulnerability detection but limited code coverage. Combining these techniques can help overcome their limitations, leading to greater scalability and precision.

Despite all the security mechanisms and safeguards of modern compilers and operating systems, software vulnerabilities still exist in released C software, i.e., binary programs. This reality highlights the importance of applying the previously mentioned techniques directly to binary programs. However, the accurate identification of a vulnerability's exploit vector in a binary is a challenging task. This is due to the disassembled binary code offering little insight into the program's higher-level logic. This leads to the need for a scalable and accurate analysis method to

[a] https://orcid.org/0009-0002-1295-2079
[b] https://orcid.org/0000-0003-4478-8680

detect vulnerabilities in binary programs. While dynamic analysis offers accuracy, it falls short in scalability for large binaries. Conversely, static analysis scales well but often lacks precision. The question then arises: *Can we devise a method that is both scalable and accurate?*

This paper proposes a static analysis approach capable of detecting BO vulnerabilities in compiled C binaries via a formal technique known as Model Checking. Model Checking is a method used to check finite state systems by exhaustively searching the system's state space to determine if some property is present. This method has been validated for verifying properties in C programs (Chen and Wagner, 2002), yet its applications to assembly code remain limited due to the state space explosion problem.

To detect BO, we propose a model checking approach to verify security properties of a binary program's stack memory. The approach involves constructing a mathematical model representing each user function's stack frame in the binary, forming the basis of the program's stack memory state space. Memory transition operators are defined to enable transitions between states within this space. To identify BO behavior, we define security properties as Linear Temporal Logic (LTL) formulas, which model proper stack memory usage. These properties are checked against the state space to identify any trace that violates them. In a positive case, we determine the vulnerability by utilizing the violated property and a counter-example trace from the Model Checker.

This paper makes the following contributions: (1) An approach using Model Checking to detect buffer overflow vulnerabilities in binary C programs; (2) Theoretical groundwork for modelling stack memory in binary programs; (3) Design of a prototype for verifying stack memory security properties.

2 BUFFER OVERFLOW VULNERABILITIES

A software vulnerability is a defect in a program that compromises the security of the program and often of the entire system it operates on (Eilam, 2005).

Currently, BOs are the most prevalent and dangerous vulnerability class (Butt et al., 2022). These vulnerabilities occur when software fails to check a buffer's bounds and writes to a memory address beyond its domain. Overflows can occur in the heap (for dynamic allocations) or stack (for local variables, function parameters, and return addresses) (One, 1996). BOs in the stack are the most harmful since a program relies on function return addresses to preserve control flow.

Listing 1 demonstrates a BO vulnerability. The code allocates and fills a 256-byte buffer (`buffer_1`, line 6) with `x`. In line 8, the `copy` function is called with `buffer_1` as the parameter. This function generates a 16-byte buffer (`buffer_2`) in line 2 and copies the contents of the first buffer using the `strcpy` function. A BO occurs because `buffer_2` is not large enough to hold the contents of `buffer_1`, leading to a spillage of excess data into adjacent memory areas.

```
1  void copy(char *str) {
2    char buffer_2[16];
3    strcpy(buffer_2, str);
4  }
5  void main() {
6    char buffer_1[256];
7    for (int i = 0; i < 255; i++)
       buffer_1[i] = 'x';
8    copy(buffer_1);
9  }
```

Listing 1: Stack Overflow Example in C

```
1  push    rbp
2  mov     rbp, rsp
3  sub     rsp, 32
4  mov     QWORD PTR [rbp-24], rdi
5  mov     rdx, QWORD PTR [rbp-24]
6  lea     rax, [rbp-16]
7  mov     rsi, rdx
8  mov     rdi, rax
9  call    strcpy
10 leave
11 ret
```

Listing 2: Copy function's x64 Assembly Code

Compiling Listing 1 to x64 Assembly enables us to investigate the memory interactions of the `copy` function, as shown in Listing 2. Initially, the base pointer (RBP) is saved on the stack with `push rbp`. To allocate space for local variables, the stack pointer (RSP) is then decremented by 32 bytes (`sub rsp, 32`). The location `RBP-24` holds an 8-byte pointer to `buffer_1`, and `RBP-16` denotes the start of a 16-byte space for `buffer_2`. When the `strcpy` function is called, it attempts to copy data from the array pointed to by `RDI` (`buffer_1`) into the space at `RBP-16` (`buffer_2`). However, since `buffer_1` contains 256 bytes, it surpasses the 16-byte limit of `buffer_2`. This causes the excess data to overflow, corrupting adjacent stack memory, including critical data such as the saved RBP and the return address of the caller function (i.e., RIP).

3 MODEL CHECKING

Model Checking is a computational technique used to analyze the behaviors of dynamic systems, represented as state-transition systems (Clarke et al.,

2017). This technique is frequently utilized to validate both hardware and software in the industry. When it is not possible to thoroughly verify the actual software, a simplified model that encompasses its core behavior can be constructed, preserving the fundamental characteristics of the system while avoiding complexities that prevent full verification. Model checking enables the verification of a system's design when directly verifying its implementation is excessively expensive. According to Clarke et al. (Clarke et al., 2017), the construction of a model checker is based on three components:

- **Model**: A Finite state-transition graph that formally describes the system, generally designated as a Kripke Structure (K).
- **Specification**: The system's desired properties are expressed using temporal logic, which is used to specify the criteria for correct state transitions.
- **Algorithms**: These computational methods check whether the state-transition model complies with the specifications in the temporal logic formulas.

The system we desire to model check is abstracted into a state-transition graph K. The specifications of the system's behavior are formulated as temporal logic formula φ. The model checker then employs a decision procedure to determine whether $K \models \varphi$ holds; in other words, it checks if K satisfies φ. Should the structure K not satisfy φ (expressed as $K \not\models \varphi$), the model checker will provide a counter-example, demonstrating how the specification φ is violated within the structure K.

4 LINEAR TEMPORAL LOGIC

Temporal logic is used to reason about the way the world changes over time. In the context of software, it is used in the specification and descriptions of systems by describing the evolution of states of a program which gives rise to descriptions of executions.

Propositional Linear Temporal Logic (LTL), as the name implies, follows the linear-time view. In addition to the operators present in propositional logic, it provides temporal operators that connect different stages of computations and talk about dependencies and relations between them (Clarke et al., 2017). Two of the most commonly used operators in LTL formulae include the following:

- $\Diamond\varphi$ (eventually): operator used to specify that a certain condition is expected to be true at some point in the future. It asserts that a future state exists in the execution where the condition holds.
- $\Box\varphi$ (always): this operator asserts that a condition must hold in all states of execution. It is used to express invariance.

To facilitate the Model Checking process, an LTL formula can be converted to a ω-automaton. These are a variation of a Finite State Automaton (FSA) that takes infinite strings as input, and, instead of having a set of accepting states, they have a variety of acceptance conditions.

The process of translating LTL requirements into ω-automaton enables the formalization of the Model Checking problem as a search for accepted runs on an automaton resulting from the synchronous product of the *State Space* and the ω-automaton (Clarke et al., 2017). Various algorithms exist for this translation, one of them is detailed in (Gastin and Oddoux, 2001).

5 STACK MODEL CHECKING APPROACH

In this paper, we propose a novel approach that aims to improve the detection of stack BO vulnerabilities. To detect these vulnerabilities, we use model checking to verify security properties within a program's stack memory. These properties model the correct usage of the stack memory space, and a violation of these would account for a potential vulnerability. Besides detecting BO vulnerabilities, the model checker also allows the verification of user-defined security properties via LTL formulas, allowing the verification of specialized properties of the stack memory.

To verify the specified security properties, we created a theoretical model of the stack memory and constructed a *state space of the program's stack*. This state space is initially constructed based on memory write operations, identified through defined transition operators. Upon completion of this construction, the model checker conducts a comprehensive search within the state space to identify any traces that violate the specified properties.

At the end of the model checking process, the violated properties and counter-example traces are emitted. If at least one security property is violated, then we determine the type of vulnerability based on the violated properties and pinpoint its source based on an analysis of the counter-example traces.

Figure 1 provides an overview of the proposed model checker's architecture, which is composed of the following modules: *Binary Data Extractor*, *Model Checker*, *Security Property Converter*, and *Vulnerability Identifier*. Next, we present an overview of these modules and their interconnections, and detailed descriptions of each module are provided in Section 6.

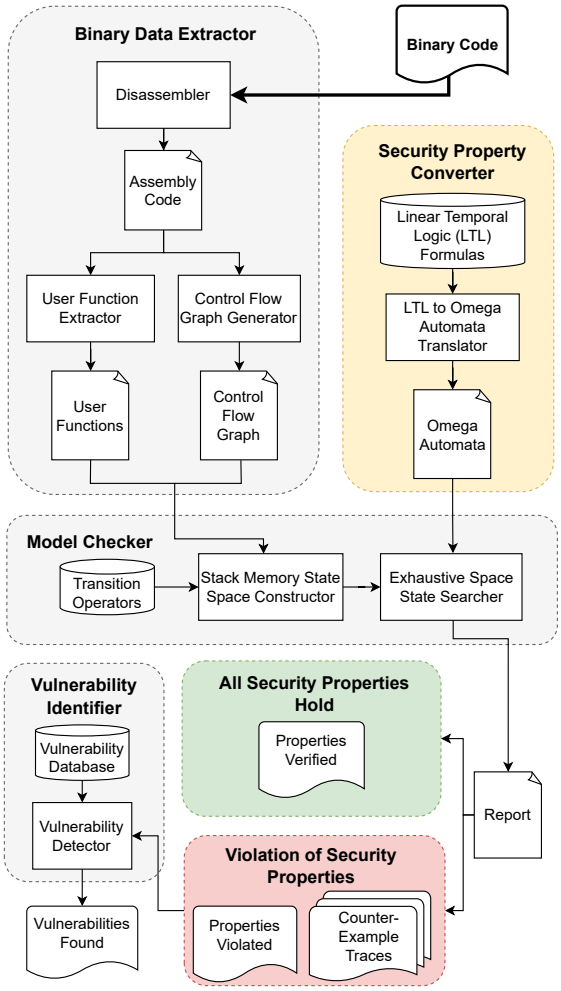

Figure 1: Overview of the model checking approach

Binary Data Extractor. This module begins by disassembling the input binary program to extract x86-64 assembly code. It performs two key analyses: (i) identifies user-defined functions, extracting function names and block addresses, and (ii) generates a control flow graph (CFG) of the program.

Model Checker. This component is in charge of building the state space and verifying security properties within the program's stack memory, and it operates in two stages:

- *Stack Memory State Space Constructor*: builds a state space model using a database of transition operators that define which assembly instructions affect the stack.
- *Exhaustive Space State Searcher*: verifies security properties, represented as omega automaton, against the program model through an exhaustive state space search.

Depending on the verification outcomes, a report is generated. If violations are detected, it includes documents listing violated properties and counter-example traces. If all properties are verified, the report lists the verified properties.

Security Property Converter. The Security Property Converter functions as an interface within our architecture, facilitating the specification of additional security properties by users. These properties are crucial for verifying a given binary, especially for customized security needs. This module stores user-specified security properties formulated as LTL formulas, alongside pre-defined properties that model the correct usage of the stack memory. These formulas are then translated to ω-automaton, before they are passed to the model checker for verification.

Vulnerability Identifier. When a security property is found to be violated, the binary is automatically forwarded to the Vulnerability Identifier. This module will attempt to pinpoint the vulnerabilities' exact source within the binary code. This process involves a two-phase approach. Initially, the type of vulnerability is determined by correlating the violated security properties with entries in a vulnerability database. Subsequently, a reverse-flow analysis of the counter-example traces is conducted to locate the precise position of the vulnerability in the program's code.

6 DESIGN INSIGHTS

This section delves into the theoretical groundwork laid for the presented approach for detecting vulnerabilities in binary programs and details the components of our architecture.

6.1 Extracting Data from the Binary

To efficiently extract all relevant data from a binary, we utilized Angr[1], an open-source binary analysis framework designed for Python, which employs the Capstone disassembly engine[2] for recursive disassembly of the binary file. This method offers enhanced accuracy in translating binary files to machine code, especially when compared to traditional linear disassemblers like objdump[3]. The framework also supports built-in analyses for extracting comprehensive data from binaries.

[1] https://angr.io/
[2] http://www.capstone-engine.org/
[3] https://linux.die.net/man/1/objdump

Utilizing Angr enables the extraction of the CFG of a binary program, along with critical User Function Data. This data encompasses essential elements such as function names and the addresses of basic blocks.

6.2 Building the Stack Memory State Space

Before implementing the Model Checker module, it was necessary to establish the foundational theoretical framework. This involved creating a model that represents the program's stack memory, which is essential for appropriately replicating and evaluating the program's memory interactions. The state space was designed to reflect different possible configurations of the stack memory as the program runs. A collection of memory operators was created to effectively control this state space. The operators specify the allowable actions on the state space, allowing the Model Checker to assess the program's behavior.

Memory State. In our approach, we define a state of the program's memory as a collection of *function stack frames*. Specifically, at any given point in the program's execution, there exists a set of active stack structures, each represented by a stack frame model of a user-defined function. Figure 2 illustrates the model for a memory state with two active function stack frames.

Function Stack Frame Model. Conceptualized as an array of bytes, this model mirrors the actual size of a function's stack frame, ensuring a one-to-one correspondence with the real stack. The unique feature of this model is that each byte in the array represents the current state of a single byte in the stack.

Each byte in the function stack frame model is characterized by one of four states – *Free*, *Critical*, *Occupied*, and *Modified* –, as outlined in the automaton in Figure 3.

State transitions are exclusively triggered by write operations, which are classified as either *risky* or *non-risky*. A *risky* write operation typically occurs when sensitive data such as return addresses or security tokens are written to the stack, causing a transition to

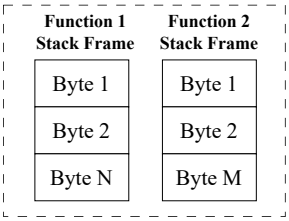

Figure 2: Conceptualized Memory State

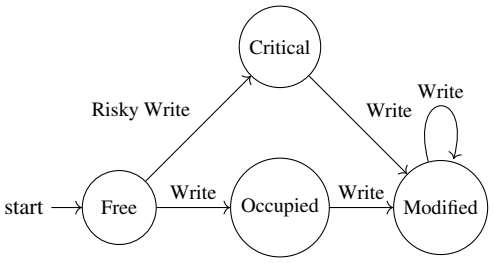

Figure 3: Automaton for the Byte States

the *Critical* state. This indicates an increased vulnerability risk at that specific stack location. *non-risky* writes, on the other hand, transition a byte to the *Occupied* or *Modified* state, depending on its prior state and the nature of the write operation. The *Free* state signifies unoccupied areas of the stack, less likely to be the target of exploitation.

Transition Operators. Constructing the state space required defining transitions between each memory state. Although the transitions were first implemented in the Byte State automaton (as depicted in Figure 3), they are more intricate. We classify them into two categories: *direct* and *indirect*.

Direct transitions are those that result from single assembly instructions directly altering the stack. Examples include instructions like `mov`, which straightforwardly modify the stack. In contrast, *indirect transitions* arise from function calls that modify the stack memory indirectly. An instance of this would be a call to the `strcpy` function, where the effect on the stack is a consequence of the function's execution rather than a direct instruction. A summarized representation of some direct and indirect memory operations is provided in Table 1

6.3 Constructing the State Space

The state space is conceptualized as a graph structure, where nodes encapsulate distinct memory states, and edges depict transitions facilitated by a predefined set of memory operations. To systematically construct this state space, we outlined algorithm 1.

We may create the state space for the assembly code of the `copy` function in Listing 2 by following Algorithm 1. This state space is illustrated in Figure 4. For the final state in this state space, we consid-

Table 1: Direct and Indirect Memory Operations.

Type of Transition	Operation
Direct	MOV
Direct	PUSH
Direct	POP
Indirect	CALL (e.g., strcpy)

Algorithm 1: Procedure to generate the state space of a binary's stack memory.

Data: CFG
Result: State Space (K)
Initialize empty K;
foreach *basic block* $B \in CFG$ **do**
  Determine the function f associated with block B;
  if *function* f *not in* K **then**
    Create new memory state with f;
  end
  foreach *instruction* $I \in B$ **do**
    Match I with memory operator;
    if *match is found* **then**
      Apply the operation to a copy of the current memory state;
      Update K with the new memory state;
    end
  end
end

ered the worst-case scenario for the `strcpy` function, when a buffer overflow occurs and overwrites every byte up to the bottom of the stack. In the absence of additional knowledge about the arguments for the function call, we consider the worst-case scenario.

6.4 Specifying and Verifying Security Properties

Our approach primarily aims to detect BO by defining security properties for our model, which represent the correct usage of the stack. Any violations of these properties indicate potential BO in the program. We utilize LTL to model these properties, supplemented by unique functions specific to our model. These functions enable referencing various model parts within LTL. For instance, we have defined the following two functions:

Definition 1. *Stack*(f)*: Given a function* f, Stack(f) *denotes the stack frame allocated for* f.

Definition 2. *Byte*(s,i)*: For a stack frame* s, Byte(s,i) *returns the current byte state for the byte at position* i *within* s.

With these, we can define our first and most critical security property (Eq. 1):

$$\square \left(\bigwedge_x Byte\left(Stack\left(x\right),0\right) = Critical \right) \qquad (1)$$

This security property expresses that the state of the first byte of each stack frame in a memory state

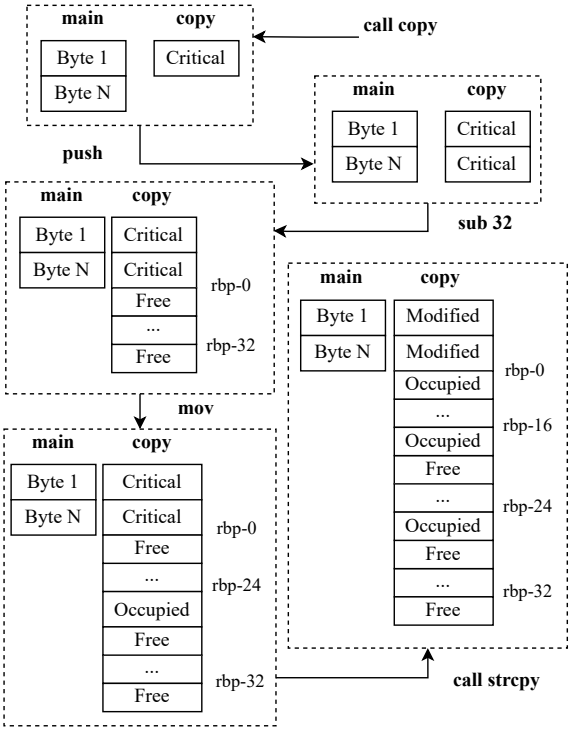

Figure 4: Simplified State Space generated from copy function's assembly code in Listing 2

must always be *Critical*. If this property is violated, it indicates that a buffer overflow has occurred and the caller's return address (i.e., RIP) was overwritten.

Before verifying this property, our model checker must convert it into an ω-automaton, a task performed by the Security Property Converter module. Using the algorithm from (Gastin and Oddoux, 2001), this module translates the properties into Büchi automaton, a specific type of ω-automaton. The Büchi automaton corresponding to Eq. 1 is present in Figure 5.

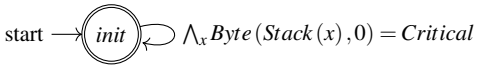

Figure 5: Automaton for the Security Property in Eq.1

Finally, to verify this property against our state space, we must find a sequence of states our automaton accepts, i.e., a sequence where the given condition holds continuously. Upon examining our state space, as shown in Figure 4, it becomes evident that in the final state, the condition of the first byte being critical is not met. This indicates that the caller's return address was overwritten, and a stack BO occurred.

6.5 Identifying Vulnerabilities

To identify vulnerabilities, our approach correlates violated properties with vulnerability classes. For in-

stance, a violation of the property detailed in Eq. 1 can indicate the presence of the vulnerability CWE-787: Out-of-bounds Write. This categorization, while general, can be refined by further defining more precise properties.

Consider the following LTL formula in Eq. 2, which ensures that no `strcpy` call is followed by an overwrite of a critical byte:

$$\neg\Big(\Diamond\Big(\bigvee_{x} Byte\,(Stack\,(x)\,,0) = Modified \wedge PreviousTransition = \text{call strcpy}\Big)\Big) \tag{2}$$

Compared to Eq. 1, this property is more specific, and a violation could indicate the presence of the vulnerability CWE-120: Buffer Copy without Checking Size of Input. However, it only accounts for `strcpy` calls and would require further generalization to encompass other function calls.

To pinpoint the vulnerability's location, we perform a reverse-flow analysis on the counter-example trace. For the property in Eq.1, this trace includes the transitions {`call copy`, `push`, `sub 32`, `mov`, `call strcpy`}, complete with their respective addresses and other pertinent details. This analysis allows us to identify the vulnerability's origin. In this case, it is determined by the address of the `call strcpy` instruction.

6.6 Preliminary Results

A seminal prototype of the Model Checker was implemented, enabling us to conduct initial tests for our approach. For evaluation, we used 10 small C programs from NIST SARD[4], each exhibiting improper use of the `strcpy` function, resulting in stack BO. Our method successfully detected violations of the security properties outlined in Eq. 1 and Eq. 2 in all cases. Consequently, it identified the presence of a CWE-120 vulnerability in each of the 10 applications.

7 RELATED WORK

Vulnerability Discovery. Vulnerability detection, a long-standing and extensively researched area, has primarily focused on source code vulnerabilities. Notable studies in this field include (Kaur and Nayyar, 2020), comparing static analysis tools for C/C++ and Java, and (Sharma et al., 2024), which reviews the use of machine learning in source code analysis.

[4]https://samate.nist.gov/SARD/

For C source code, (Inácio and Medeiros, 2023) introduces a tool that integrates static and dynamic analysis to detect and automatically repair buffer overflow vulnerabilities. It employs static analysis to identify potential overflows and extracts corresponding code slices. These slices are compiled and subjected to fuzzing, allowing the validation of vulnerabilities as either true or possible false positives.

Identifying vulnerabilities in binary code presents greater challenges than source code due to information loss in compilation. However, significant efforts like (Vadayath et al., 2022) have been made. In this work, the authors combine static and dynamic analysis to detect vulnerabilities in binary files. Their tool, Arbiter, is particularly effective at identifying key vulnerabilities, such as Incorrect Calculation of Buffer Size and Uncontrolled Format String, among others.

The most common Dynamic Analysis technique for discovering vulnerabilities is Fuzzing. This approach involves creating test cases, typically using odd inputs, to intentionally crash a program and identify potential problems. The study conducted by (Li et al., 2018) examines the latest developments in fuzzing solutions. In a separate advancement, the technique of grey-box concolic testing for test case generation was introduced by (Choi et al., 2019).

Model Checking in Software Security. Model Checking is traditionally used to model and study the behavior of software and hardware, typically emphasizing the validation of certain functionalities or the absence of unwanted behaviors.

The direct application of these techniques for vulnerability discovery is uncommon, however some research has been done with this purpose. For web security, (Huang et al., 2004) used bounded model checking to verify the source code of web applications.

To Verify C code, some tools exist in the literature, most notably (Chen and Wagner, 2002), which presented MOPS, a tool to examine security properties in C software. A different tool for validating C source code was introduced by Kroening et al. (Kroening and Tautschnig, 2014), which uses bounded model checking to verify memory safety features.

Although not commonly used in binary code due to the state explosion problem, model checking has been used to detect malware behaviors, and validate micro-controller code (Mercer and Jones, 2005). Notably, Nguyen et al. (Nguyen and Touili, 2017) developed SPCARET, a temporal logic to detect malware.

For exploit discovery, (Eckert et al., 2018) developed a framework, HeapHopper, based on bounded model checking and framework execution, to analyze the exploitability of different heap implementations.

Unlike these works, we propose a novel approach for vulnerability detection. Although we also resort to model checking, we propose its use differently. We construct the stack memory state space of binary programs and use model checking to verify security property violations over it.

8 CONCLUSIONS

In this paper, we introduced a model checking approach for binary programs, aimed at detecting stack BO vulnerabilities by verifying security properties of the stack memory. Our proposal includes developing a theoretical framework for modelling stack memory and formulating security properties for its analysis. Future improvements should focus on increasing the precision of state space generation to better identify various stack BO vulnerabilities and expanding our security properties to cover complex malicious behaviors such as return-oriented programming (ROP).

As the next steps, we plan to advance our model by adding new security properties and enhancing LTL formulas with additional predicates, aiming to improve stack BO detection and categorization. We will then evaluate our approach with diverse binaries of C applications, focusing on the precision in vulnerability detection and the scalability of our approach. Based on the evaluation outcomes, we may adjust the model and properties to enhance performance.

ACKNOWLEDGMENTS. This work was supported by FCT through the LASIGE Research Unit, ref. UIDB/00408/2020 (https://doi.org/10.54499/UIDB/00408/2020) and ref. UIDP/00408/2020 (https://doi.org/10.54499/UIDP/00408/2020)

REFERENCES

Butt, M. A., Ajmal, Z., Khan, Z. I., Idrees, M., and Javed, Y. (2022). An in-depth survey of bypassing buffer overflow mitigation techniques. *Applied Sciences*, 12(13).

Chen, H. and Wagner, D. (2002). Mops: an infrastructure for examining security properties of software. *Proceedings of the ACM Conference on Computer and Communications Security*.

Choi, J., Jang, J., Han, C., and Cha, S. K. (2019). Grey-box concolic testing on binary code. In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 736–747.

Clarke, E. M., Henzinger, T. A., Veith, H., and Bloem, R., editors (2017). *Handbook of Model Checking*. Springer Cham.

Eckert, M., Bianchi, A., Wang, R., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. (2018). Heaphopper: bringing bounded model checking to heap implementation security. In *Proceedings of the USENIX Security Symposium*, page 99–116.

Eilam, E. (2005). *Reversing: Secrets of Reverse Engineering*. John Wiley & Sons, Inc., USA.

Gastin, P. and Oddoux, D. (2001). Fast ltl to büchi automata translation. In *Comp. Aided Verification*, pages 53–65.

Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D., and Kuo, S.-Y. (2004). Verifying web applications using bounded model checking. In *International Conference on Dependable Systems and Networks, 2004*, pages 199 – 208.

Inácio, J. and Medeiros, I. (2023). Corca: An automatic program repair tool for checking and removing effectively c flaws. In *IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 71–82.

Kaur, A. and Nayyar, R. (2020). A comparative study of static code analysis tools for vulnerability detection in c/c++ and java source code. *Procedia Computer Science*, 171:2023–2029.

Kroening, D. and Tautschnig, M. (2014). Cbmc – c bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413, pages 389–391.

Kroes, T., Koning, K., Kouwe, E., Bos, H., and Giuffrida, C. (2018). Delta pointers: buffer overflow checks without the checks. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–14.

Li, J., Zhao, B., and Zhang, C. (2018). Fuzzing: a survey. *Cybersecurity*, 1.

Mercer, E. and Jones, M. (2005). Model checking machine code with the gnu debugger. In *Proceedings of the 12th International Conference on Model Checking Software*, page 251–265.

Nadeem, M., Williams, B. J., and Allen, E. B. (2012). High false positive detection of security vulnerabilities: a case study. In *Proceedings of the 50th Annual Southeast Regional Conference*, page 359–360.

Nguyen, H.-V. and Touili, T. (2017). Caret model checking for malware detection. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, page 152–161.

One, A. (1996). Smashing the stack for fun and profit. *Phrack*, 7(49).

Sharma, T., Kechagia, M., Georgiou, S., Tiwari, R., Vats, I., Moazen, H., and Sarro, F. (2024). A survey on machine learning techniques applied to source code. *Journal of Systems and Software*, 209:111934.

Vadayath, J., Eckert, M., Zeng, K., Weideman, N., Menon, G., Fratantonio, Y., Balzarotti, D., Doupé, A., Bao, T., Wang, R., Hauser, C., and Shoshitaishvili, Y. (2022). Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs. In *Proc. of the USENIX Security Symposium*, pages 413–430.