Computers & Security 172 (2027) 105165

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Full length article

BASICS: Binary Analysis and Stack Integrity Checker System for Buffer Overflow Mitigation

Luís Ferreirinha [a], Ibéria Medeiros [b,*]

[a] VUSec, Vrije Universiteit Amsterdam, De Boelelaan, Amsterdam, 1081 HV, Amsterdam, The Netherlands
[b] LASIGE, Faculty of Sciences of the University of Lisboa, Campo Grande, Lisboa, 1749-016, Lisboa, Portugal

ARTICLE INFO

Keywords:
Buffer overflow detection
Binary code
Model checking
Concolic execution
Binary patching
Software security

ABSTRACT

Linux-based systems rely on large ecosystems of packaged binaries, many of which are implemented in C/C++. While these languages provide low-level memory control and high performance, they are also highly susceptible to memory-safety vulnerabilities, particularly buffer overflows (BOs). Traditional vulnerability discovery techniques, such as static and dynamic analysis, often struggle with scalability and precision when applied directly to binary code, which can thereby keep programs vulnerable. This work introduces BASICS, an approach and framework that combines model checking and bounded concolic execution to automatically verify security properties of stack memory in binary code, trampoline-based binary repair and crash-inducing inputs for bounded patch validation. BASICS constructs a *Memory State Space* – `MemStaCe`– from the binary program's control-flow graph and bounded concolic execution of C function calls and loops. Security properties, defined in Linear Temporal Logic, model vulnerable stack behaviours and allow the approach to identify violations in `MemStaCe` through counterexample traces. These vulnerabilities are then addressed using trampoline-based binary patching, and the resulting patches are validated using crash-inducing inputs extracted during concolic execution. We evaluated BASICS on the Juliet C/C++ and SARD benchmarks, as well as real open-source and Linux package binaries. BASICS achieved 86.4% precision and 87.4% accuracy on SARD, and 86.1% precision on Juliet, while also generating validated binary patches for supported vulnerable sinks. Compared with other evaluated tools, it provided the strongest overall balance between precision, analysis cost, and patching capability.

1. Introduction

Linux-based systems are deployed across a wide range of platforms, from embedded devices and personal desktop computers to enterprise servers. The Linux kernel forms the core of these systems, while the surrounding user-space environment is composed of large ecosystems of packaged binaries, including system utilities, long-running services, third-party applications, and embedded user-space components (Debian Project, 2026; The Linux Kernel Organization, 2025). A significant fraction of this software is implemented in C/C++, where low-level memory manipulation remains common. Although this programming model gives developers precise control over data layout, performance, and interaction with the operating system, it also exposes programs to memory-safety vulnerabilities (Szekeres et al., 2013). Stack-based buffer overflows (BOs) are a particularly important class of such vulnerabilities: an out-of-bounds write to a stack object can corrupt adjacent local variables, saved frame pointers, return addresses, or other control-relevant data, potentially enabling crashes, control-flow hijacking, or arbitrary code execution (MITRE Corporation - Common Weakness Enumeration, 2025a).

Despite widespread compiler and operating-system mitigations, including stack canaries, address-space layout randomisation, non-executable stacks, and control flow hardening, BOs have not disappeared from native software deployed (Shacham et al., 2004; Abadi et al., 2005; Cowan et al., 1998; CybersecurityDive, 2025) and remain ranked by CWE as one of the most dangerous vulnerabilities (Department, 2025). In practice, hardening is not uniform across all binaries, and some protections depend on compiler options, linker options, source-level patterns, or deployment-specific build configurations (Open Source Security Foundation, 2026). Consequently, security analysts often need to reason directly about binaries: they must detect whether a compiled program contains a vulnerable stack write, identify where the vulnerability occurs, and, when source-level repair is unavailable or impractical, apply a mitigation directly to the binary.

Detecting vulnerabilities such as BOs has been a long-standing problem, with a plethora of different tools and methods developed (Kroes

* Corresponding author.
E-mail address: ivmedeiros@fc.ul.pt (I. Medeiros).

https://doi.org/10.1016/j.cose.2026.105165
Received 20 February 2026; Received in revised form 5 September 2026; Accepted 9 September 2026
Available online 30 September 2026
0167-4048/© 2026 Elsevier Ltd. All rights are reserved, including those for text and data mining, AI training, and similar technologies.

et al., 2018; Inácio and Medeiros, 2023; Wang et al., 2021; Russell et al., 2018; Liu et al., 2019; Xu and Wang, 2022). These tools employ various analysis methods, but often follow one of the following approaches: Static Analysis, Dynamic Analysis, or a hybrid of both. Static analysis analyses a program's code without executing it, achieving high code coverage at the cost of increased false positives (Meng and Miller, 2016). In contrast, dynamic analysis techniques execute the code of a program (Ball, 1999), allowing for more accurate vulnerability detection, but at the cost of code coverage. Hybrid approaches have been proposed to mitigate these limitations (Inácio and Medeiros, 2023; Aggarwal and Jalote, 2006; Kim et al., 2015; Vadayath et al., 2022). Although improvements were achieved, these techniques often struggle with scalability and precision when applied to binary code, still leaving programs vulnerable.

In addition to this detection deficit, these vulnerabilities, when discovered, are generally reported to developers, whose job is to remove them. However, developers cannot always provide an immediate patch, which often results in a prolonged vulnerability life of up to 12 months in some cases (Hu et al., 2019), thus keeping an open window for attackers to continue exploiting the systems that contain them. Research on this issue has led to the development of methods that automatically patch vulnerabilities. Although most approaches focus on source code (Inácio and Medeiros, 2023; Wang et al., 2021; Lin et al., 2007), recent work has expanded to binary-level patching. Tools such as E9Patch (Duck et al., 2020) allow users to replace sections of a binary with custom patches, and other works aim to completely automate this process (Duan et al., 2019; Hu et al., 2019). However, to effectively remove these vulnerabilities, one must first identify their location within the binary, a challenging task with disassembled programs that offer little insight into the higher-level logic present in the source code. Then, as a second step, the validation of such patches involves checking that they effectively remove the target vulnerabilities without introducing new ones and compromising the program's correct functioning. Therefore, a significant gap can then be identified in the current landscape of vulnerability mitigation tools: *the absence of a scalable, accurate, and automated solution to identify and patch vulnerabilities in binary programs*.

This paper presents a novel approach to automatically detect and mitigate BOs in compiled C binaries. We call this approach *Binary Analysis and Stack Integrity Checker System* (BASICS). BASICS combines static binary analysis, bounded symbolic execution, and model checking. Static analysis recovers the binary-level control-flow structure, bounded symbolic execution refines the stack effects of calls and bounded loops, and model checking verifies Linear Temporal Logic (LTL)-specified stack-security properties over the resulting stack memory model. When a violation is found, a counterexample is produced and used to localise the vulnerable sink and guide trampoline-based binary patching, followed by bounded validation of the patched binary.

The most relevant works on software model checking in the literature tend to focus on verifying models of distributed systems (Holzmann, 1997), or on bounded verification of source-level C programs (Clarke et al., 2004). Although these do not directly detect vulnerabilities, they can be used to discover design flaws in programs that might lead to a vulnerability. Moreover, the use of symbolic and concolic methods have been widely used to improve vulnerability discovery, often in combination with fuzzing execution (Yun et al., 2018; Chen et al., 2022; Mouzarani et al., 2017). Such approaches typically search for crashing inputs or path witnesses rather than constructing a temporal memory model for checking explicit stack-integrity properties. Finally, automatic program repair has mainly focused on source-level patches (Monperrus, 2018), whereas binary-level repair remains more challenging because source-level structure and type information are unavailable. The proposed approach addresses this gap by constructing a binary-level memory-state model, checking LTL stack-security properties over that model, and applying validated trampoline-based patches directly to the binary.

The approach models the stack memory of the binary program as a *Memory State Space* (`MemStaCe`). It constructs this state space from the binary control-flow graph and refines it with bounded symbolic execution, which is used to approximate the stack effects of selected C library calls, indirect data-dependent writes, and bounded loop executions. In `MemStaCe`, each transition between memory states corresponds to an operation in the stack memory (e.g., pushing a register to the stack). Afterwards, `MemStaCe` is used to check whether security properties are violated, thus flagging the potential existence of BOs and generating a counterexample trace with the instructions involved in each violation. Security properties model the correct usage of the stack memory space of functions related to BO vulnerabilities (e.g., `strcpy`). These are defined in Linear Temporal Logic (LTL) (Clarke et al., 2018) and translated into Büchi-Automaton before being used by the model checker in the BO discovery task. Traces are submitted to a reverse-flow analysis to pinpoint the location of the vulnerability sink (a function like `strcpy` that is sensitive to malicious inputs), and then patched with binary fixes (small binary programs containing the correct code that fix the vulnerability) to remove the found BOs by using a trampoline-based binary patching method. Lastly, the approach checks the effectiveness of the patches using vulnerable concolic inputs derived from the symbolic execution process, alongside benign and boundary inputs.

The paper also presents BASICS's implementation for detecting and removing BOs, with its effectiveness validation. We evaluated BASICS based on three different criteria. First, we assessed its ability to detect BO vulnerabilities in binaries using the Juliet C/C++ test suite and a subset of the NIST SARD (NIST, 2021) dataset and compared its effectiveness against other tools, achieving the strongest overall balance between precision, analysis cost, and patching capabilities. Second, we evaluated how BASICS is effective in mitigating BOs with the SARD subset. Finally, we evaluated the scalability of BASICS, by performing a sweep of 292 binaries from Linux repositories and 3 large open-source projects.

The main contributions of the paper are: (1) a novel binary-level approach that combines static analysis, bounded symbolic execution, model checking, and trampoline-based patching to detect, localise, and mitigate stack-based BO vulnerabilities; (2) a Memory State Space (`MemStaCe`) abstraction for representing binary-level stack behaviour, constructed from control-flow recovery, stack-memory modelling, and bounded symbolic exploration of calls and loops; (3) a framework to define security properties to model the correct usage of `MemStaCe`, resorting to Linear Temporal Logic (LTL); (4) a trampoline-based binary patching method, including patch templates, to remove BOs and validate the correction; (5) the BASICS tool (available at Ferreirinha and Medeiros (2026)) that implements the main approach, including the aforementioned methods; (6) an evaluation that demonstrates the ability of BASICS in the detection and removal of vulnerabilities.

The remaining of the paper is organised as follows. Section 2 details the background that supports our approach, Sections Section 3 to 7 present in detail the approach itself, how it works, and its implementation, Section 8 evaluates the approach, Section 9 its limitations, and Sections 10, 11 and 12 present, respectively, how BASICS can be extended to other vulnerabilities, related work, and conclusions.

2. Concepts

This section presents the key concepts necessary for a deeper understanding of our approach. Firstly, it provides an overview of a BO vulnerability, the type of vulnerability we focus on throughout the paper's explanations, and then the techniques behind our approach — model checking, linear temporal logic (LTL), and concolic execution.

2.1. Stack buffer overflows

Languages such as C/C++ do not have built-in safeguards to prevent the programmer from accessing memory outside the bounds of buffers (a continuous, fixed-sized region of the stack memory used to store local variables, function parameters, and function return addresses (Szekeres et al., 2013)). As a result, out-of-bounds memory operations can occur, such as writing beyond the upper boundary of a buffer and overwriting function return addresses, which are essential to maintain the intended control flow of the program. These lead to a weakness known as a buffer overflow (BO) vulnerability. Depending on the nature of the overflow, they might be benign or cause a program to crash outright, but in some cases an attacker can influence the buffer's contents and exploit it, redirecting execution to malicious code.

```
1 void copy(char *str) {
2     char buffer_2[16];
3     strcpy(buffer_2, str);
4 }
5
6 void main() {
7     char buffer_1[256];
8
9     for (int i = 0; i < 255; i++)
10        buffer_1[i] = 'x';
11    copy(buffer_1);
12 }
```

Listing 1: Stack buffer overflow vulnerability example in C.

The code in Listing 1 demonstrates a standard example of a BO vulnerability. In this code snippet, a 256-byte buffer (`buffer_1` on line 7) is allocated and filled with the character `x` (lines 9–10). Subsequently, the `copy` function (line 11) is invoked with this buffer as a parameter to copy its content to a 16-byte buffer (`buffer_2` on line 2), calling the `strcpy` function from the C standard library. The `strcpy` function is considered dangerous as it does not take into account the size of the destination buffer when copying contents between buffers. Because `buffer_2` is not large enough to accommodate the data from `buffer_1`, this operation results in a BO, where excess data spills into the adjacent memory space.

By compiling the previous code to Assembly x86-64, we gain the ability to analyse the inner workings of the `copy` function and its interactions with memory. The code in Listing 2 shows this assembly code of the stack frame for the `copy` function call. Initially, the register base pointer (RBP) of the previous function (`main`) is preserved on the stack with the instruction `push rbp`, leaving the register RBP free to receive the current value of the register stack pointer (RSP), denoted by the instruction `mov rbp, rsp`, and thus stating the stack frame base for the `copy` function. Afterwards, RSP decreases by 32 bytes with `sub rsp, 32`, allocating space for the local variables of the `copy` function in its stack frame. Within this space, an 8-byte pointer to `buffer_1` is stored at the address RBP−24 (line 4), provided by the RDI register when the function `copy` is called. Furthermore, the address of the `buffer_2` located at the RBP−16 (line 6) is assigned to the RAX register. Then, lines 7 and 8 prepare the arguments for invoking the `strcpy` function. When `strcpy` is invoked, it is instructed to copy data from the location pointed to by RDI (line 8, which currently is `buffer_1`) to the space starting at the address RBP−16 (represented by RSI and indicating where `buffer_2` starts). Since `buffer_1` contains 256 bytes of data, it far exceeds the 16-byte capacity of `buffer_2`. Consequently, excess data from `buffer_1` overflows and corrupts the adjacent memory space beyond `buffer_2`, thus overflowing the control data stored in the stack, namely RBP and RIP (register instruction pointer), the last being the pointer of the caller function (line 11 of Listing 1).

```
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 nop
11 leave
12 ret
```

Listing 2: `copy` function's x86-64 assembly code.

2.2. Model checking

Model Checking is a computational technique used to analyse the behaviours of dynamic systems, which are represented as state transition systems (Clarke et al., 2018). This model retains the system's essential properties and allows for the verification of a system's design when its complete implementation is too complex to verify directly.

A model checker can be described as being composed of three main components (Clarke et al., 2018): **Model**: A finite state-transition graph that provides adequate formalism for the description of the system, generally designated as a Kripke Structure, denoted as K; **Specification**: The system's desired properties are expressed as temporal logic formulas φ, which provide a framework for specifying the correctness criteria of state transitions, i.e., the system's behaviour; **Algorithm**: A computational method used to determine whether the state transition model follows the specifications outlined in the temporal logic formulae.

Together, these components facilitate the model checking process. The model checker employs a decision procedure to determine whether $K \vDash \varphi$ holds, i.e., if the Kripke structure K satisfies the property φ. Should K not satisfy φ (expressed as $K \nvDash \varphi$), the model checker provides a counterexample (i.e., a correctness proof), demonstrating how the security property φ is violated within the structure K.

2.3. 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 description of systems by describing the evolution of states of a program, which gives rise to descriptions of executions (Clarke et al., 2018).

Linear Temporal Logic (LTL), as the name implies, follows the linear-time progression view. In addition to the operators present in propositional logic, this logic provides temporal operators that connect different stages of the computations and represent dependencies and relations between them. LTL formulas are constructed using normal Boolean operators ($\neg$, $\vee$, $\wedge$) and the temporal operators *next*, *previous*, *until* and *since*. These operators can then be used to define temporal abbreviations, with the most commonly used operators in LTL formulae being: $\Diamond\varphi$ (eventually), $\Box\psi$ (always), $\varphi\mathcal{W}\psi$ (Weak-Until), $\varphi\mathcal{R}\psi$ (Release) (Clarke et al., 2018).

To facilitate the model checking process, an LTL formula can be converted to an ω-automaton, thus enabling the formalisation of the model checking problem as a search for accepted runs on the synchronous product of the State Space and a ω-automaton (Gerth et al., 1995).

2.4. Concolic execution

Concolic Execution is a method that combines symbolic and concrete execution, meaning that symbolic and concrete values are used for

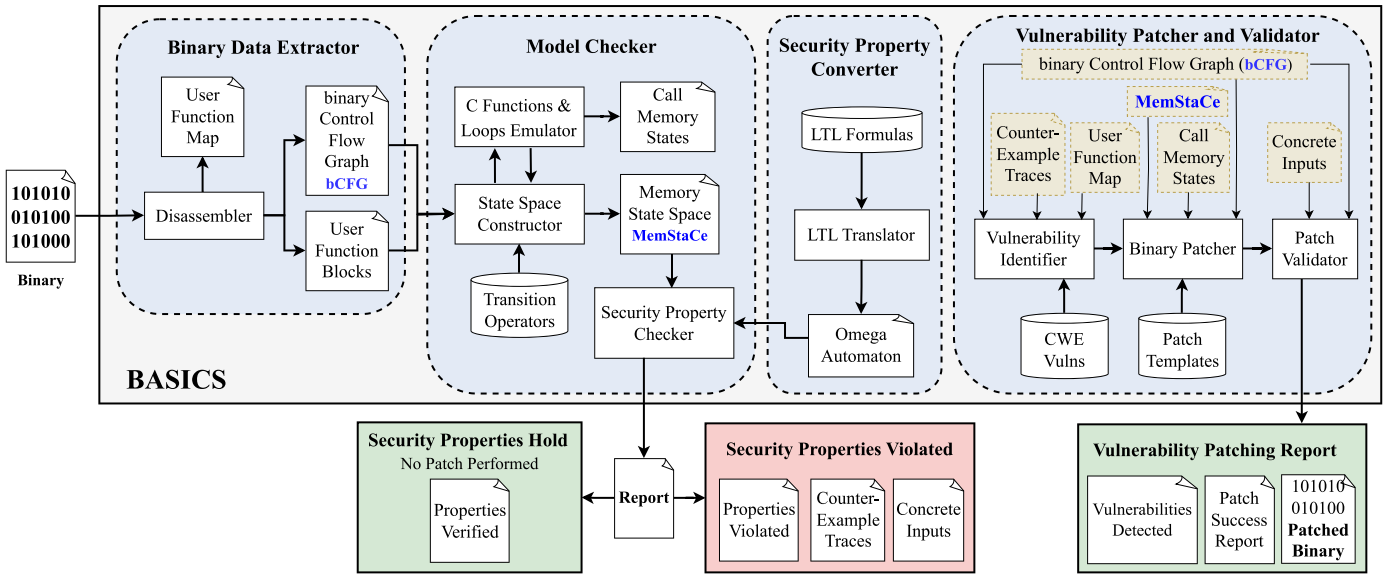

Fig. 1. Overview of the proposed BASICS's approach.

inputs, and the program is executed both symbolically and concretely. The concrete execution part constitutes the normal execution of the program, while the symbolic execution collects symbolic constraints over the symbolic input values at each branch point encountered along the concrete execution path (Sen, 2007). The process starts with the execution of the program on a set of initial inputs. As the program runs, it collects constraints on the inputs from conditional statements encountered along the execution path. These constraints are then used to generate a symbolic representation of the program execution, capturing the relationships between inputs and the program's behaviour. To solve these constraints and determine whether a path is executable, Satisfiability Modulo Theories (SMT) Solvers (De Moura and Bjørner, 2008) are used.

3. Basics approach overview

BASICS's approach analyses binary programs to detect stack-buffer-overflow vulnerabilities and repair them, including correctness validation. The approach first constructs a *Memory State Space* (`MemStaCe`), an abstract transition system that records program locations together with stack configurations. The `MemStaCe` captures direct stack effects of assembly instructions and incorporates summary transitions for external C library calls and loops, whose effects are recovered through static argument recovery and bounded concolic execution.

Security properties are expressed in Linear Temporal Logic (LTL) over observations of `MemStaCe` states, such as stack-frame metadata, byte states, recovered buffer boundaries, and the previous transition. These properties are translated to automata and checked over the `MemStaCe`. When a property is violated, BASICS emits a counterexample trace that identifies the sequence of binary instructions and abstract memory actions that lead to the unsafe stack state. This trace is used to classify the finding and, when the violation is linked to a supported call sink, to conduct binary patch generation.

For patchable findings, BASICS selects a patch template for the vulnerable sink, instantiates it using recovered call-state information such as argument registers, destination-buffer bounds, stack-canary metadata, and continuation addresses, and installs the payload into the binary using E9Patch. The patched binary is then validated under bounded executions using malicious, benign, and boundary inputs. This validation provides evidence that the observed vulnerable behaviour is remediated and that tested benign and boundary behaviours are preserved.

As shown in Fig. 1, the approach consists of four modules: *Binary Data Extractor*, *Model Checker*, *Security Property Converter*, and *Vulnerability Patcher and Validator*. We provide a quick overview of these next.

Binary Data Extractor. This module is responsible for disassembling a binary program, extracting its binary Control Flow Graph (bCFG), the User Functions Definition it contains, including function names and starting addresses, and the User Function Address Map that will serve as a reference for later vulnerability detection and patching. Similarly to a CFG extracted from the source code, the bCFG contains the basic blocks representing assembly instructions and the branches derived from control-transfer instructions (e.g., `jump`). We derive user-function information by iterating over the basic blocks of bCFG and identifying the entry address of each function, i.e., the start address of its defining block.

Model Checker. This module constructs the `MemStaCe` by traversing the recovered bCFG and translating stack-relevant instructions into abstract memory actions. It models direct stack updates, such as writes, pushes, pops, frame extensions, frame activation, buffer metadata, and canary metadata. It also incorporates summary transitions for external calls and loops. These summaries are computed using recovered call arguments and bounded symbolic execution, and may produce concrete inputs for later validation. The model checker then evaluates the automata generated from the LTL properties over the `MemStaCe` and emits one of two reports: *Security Properties Violated*, containing the list of the violated properties, the corresponding counterexample traces, and the concreted inputs; or *Security Properties Hold*, containing the properties satisfied by the explored `MemStaCe`.

Security Property Converter. This module translates LTL security properties into automata used during model checking. The properties specify unsafe stack behaviours, such as modification of control-sensitive bytes, use of unsafe library calls, and writes that cross recovered stack-buffer boundaries. The resulting automata are checked against the property-specific Kripke structure derived from the `MemStaCe`.

Vulnerability Patcher and Validator. This module consumes violated properties and their counterexample traces. It first identifies whether the finding is associated with a supported patchable sink, currently external-library call sites such as `strcpy`, `gets`, `sprintf`, `memcpy`, and related functions. If the finding is patchable, the module selects and

instantiates a patch template, installs the generated payload into the binary using E9Patch, and records patch installation evidence. Finally, it validates the patched binary under bounded malicious, benign, and boundary executions, reporting remediation, preservation, failure, or inconclusive outcomes.

The following sections describe each stage in detail. Section 4 defines the construction of MemStaCe, Section 5 explains how LTL properties are checked and how violations become vulnerability reports, and Section 6 presents patch generation and bounded validation. The implementation of these modules in BASICS is described in Section 7.

4. Constructing the memory state space

This section presents our proposed memory state space for binaries (MemStaCe), starting with the formal definition of its components, and culminating in its definition, followed by the construction of external calls and loops.

4.1. Formal definition of MemStaCe

The model checker constructs the *Memory State Space* (MemStaCe) of a binary program and explores it to detect stack-memory states that may violate security properties related to buffer overflows. Formally, MemStaCe is constructed in several stages. First, BASICS recovers a binary control-flow graph (bCFG) that captures the possible execution paths of the program. Second, it defines an abstract representation of stack memory consisting of stack frames, buffer metadata, and security-relevant byte states. Third, assembly instructions are mapped to abstract memory actions that model stack updates independently of the underlying instruction set. Fourth, operational semantics are defined for these actions, describing how they transform abstract stack configurations. Finally, the resulting stack transitions are combined with control-flow transitions to produce MemStaCe, a transition system whose states represent both the current program location and the corresponding abstract stack state.

Binary Control Flow Graph. For a binary program P, let

$$bCFG_P = (V, E, \rho_0, \mathsf{inst})$$

be the binary control-flow graph recovered by BASICS. The set V contains program locations, $E \subseteq V \times V$ is the control-flow edge relation, and $\rho_0 \in V$ is the entry location. If $(\rho, \rho') \in E$, execution may flow from ρ to ρ'. The map

$$\mathsf{inst} : V \to \mathtt{Instr}$$

assigns an instruction to each program location. When the recovered bCFG contains basic-block nodes, we assume the graph is normalised so that each node contains at most one memory-relevant instruction. Equivalently, a basic block with multiple memory-relevant instructions is split into a sequence of program locations.

Stack Model. Let Stack denote the set of all abstract stack configurations. A stack configuration $K \in \mathsf{Stack}$ is an ordered sequence of stack frames:

$$K = F_0 \cdot F_1 \cdots F_n, \qquad \mathrm{top}(K) = F_n$$

A single execution path has one current stack configuration at each program location. MemStaCe may contain many such configurations because different control-flow paths can produce different stacks. The order is significant: F_n is the currently active frame, while $F_0, \ldots, F_{n-1}$ are caller frames.

Before we define a frame, we introduce buffer descriptors and byte-states. For a given frame with p recorded stack buffers, the buffer metadata is a set of buffer descriptors σ

$$\Sigma : \{\sigma_0, \ldots, \sigma_{p-1}\}$$

$$a \in Act \quad ::= \quad \mathrm{ActivateFrame}(\ell) \mid \mathrm{Write}(o, w, bt) \mid \mathrm{Extend}(n) \mid \mathrm{Push}(n, bt) \mid \mathrm{Pop}(n) \mid \mathrm{Buf}(\sigma) \mid \mathrm{ExtCall}(c, \beta) \mid \mathrm{Loop}(L, \beta) \mid \mathrm{Canary} \mid \epsilon.$$

Fig. 2. Abstract memory actions.

$$bt \in BTrans ::= \mathrm{nRWrite} \mid \mathrm{RWrite}$$

Byte transition. $\boxed{\delta_{bt} : BState \to BState}$

$$\delta_{\mathrm{nRWrite}}(q) \triangleq \begin{cases} \mathrm{Occupied} & q = \mathrm{Free}, \\ \mathrm{Modified} & q \in \{\mathrm{Occupied}, \mathrm{Critical}, \mathrm{Modified}\}, \end{cases}$$

$$\delta_{\mathrm{RWrite}}(q) \triangleq \begin{cases} \mathrm{Critical} & q = \mathrm{Free}, \\ \mathrm{Modified} & q \in \{\mathrm{Occupied}, \mathrm{Critical}, \mathrm{Modified}\}. \end{cases}$$

Byte-array update. $\boxed{B[o, w \leftarrow bt]}$

$$B[o, w \leftarrow bt] \triangleq B' \quad \text{where} \quad \mathrm{dom}(B') = \mathrm{dom}(B)$$

$$\text{and} \quad \forall i \in \mathrm{dom}(B).\ B'[i] = \begin{cases} \delta_{bt}(B[i]) & o \le i < o + w, \\ B[i] & \text{otherwise.} \end{cases}$$

Fig. 3. Byte transitions and byte-array update semantics.

A buffer descriptor is a pair $\sigma = (o, n)$, where o is the starting offset (i.e., $\mathrm{start}(\sigma) = o$) and n is the size of the buffer; so $\mathrm{end}(\sigma) = o + n - 1$.

Additionally, for a frame with m modelled bytes, the byte-state array is a finite map

$$B : \{0, \ldots, m-1\} \to BState,$$

with possible byte states being

$$bs \in BState ::= \mathrm{Free} \mid \mathrm{Occupied} \mid \mathrm{Critical} \mid \mathrm{Modified}$$

The state Free denotes reserved but uninitialised stack space; Occupied denotes ordinary initialised stack data; Critical denotes control or security-sensitive data, such as saved return addresses or frame pointers; and Modified denotes a byte that has been overwritten after initialisation.

If a stack canary is detected for a frame, its presence is recorded by the Boolean metadata field γ:

$$\gamma \in \mathbb{B} ::= \mathsf{true} \mid \mathsf{false}.$$

Finally, we define a frame as a tuple that records a function label, a byte-state array, buffer metadata, and canary metadata:

$$F = (\ell, B, \Sigma, \gamma).$$

Abstract Memory Actions. MemStaCe does not give semantics directly to assembly instructions. Instead, each instruction is first translated into an abstract memory action defined in Fig. 2.

The action $\mathrm{ActivateFrame}(\ell)$ makes the frame associated with function label ℓ the active frame. If no such frame exists in the current stack configuration, it creates a fresh frame for ℓ. Otherwise, it moves the existing frame for ℓ to the top of the stack. The action $\mathrm{Write}(o, w, bt)$ updates w bytes starting at offset o, according to the specified *byte-transition*, bt. These are defined in Fig. 3, and will be discussed later alongside the semantics of the actions. The actions $\mathrm{Extend}(n)$, $\mathrm{Push}(n, bt)$, and $\mathrm{Pop}(n)$, respectively, allocate, append, and remove bytes from the current frame. The action $\mathrm{Buf}(\sigma)$ records buffer metadata. Calls and

loops are represented by $\mathrm{ExtCall}(c, \beta)$ and $\mathrm{Loop}(L, \beta)$, where c identifies the program location ρ at which the external function call occurs, L identifies the program location ρ of the loop header or entry point, and β represents the bounds used during bounded symbolic execution. The action Canary denotes the presence of a canary in the stack frame, and the action ϵ symbolises an instruction with no relevant stack-memory effect.

We write

$$\alpha : V \to Act$$

for the labelling function that maps each program location to its corresponding abstract memory action:

$$\alpha(\rho) \triangleq \mathrm{MemAct}(\mathsf{inst}(\rho)),$$

where MemAct : Instr → *Act* abstracts concrete instructions into `MemStaCe` actions.

Semantics. The semantics of memory actions are defined in four layers. First, byte transitions describe how individual byte states evolve when a write occurs. Second, stack operations define changes to the active stack-frame context. Third, frame operations lift byte transitions to individual stack frames. Finally, action semantics lifts stack and frame operations to complete stack configurations and defines how `MemStaCe` transitions follow the bCFG.

Fig. 3 defines the byte-level semantics. The transition type bt distinguishes ordinary non-risky stack writes, represented by nRWrite, from writes that initialise control-sensitive stack data, represented by RWrite. A non-risky write changes a Free byte into Occupied, while a risky write changes a Free byte into Critical. In both cases, writing to a byte that was already Occupied, Critical, or Modified changes it to Modified. Thus, the Modified state records that a previously initialised byte has been written again.

The byte-array update $B[o, w \leftarrow bt]$ applies the corresponding byte transition to all bytes in the interval $[o, o + w)$, while leaving all other bytes unchanged. The domain of the byte array is preserved by the update; the validity of the accessed interval is enforced by the frame-level write rule.

Fig. 4 defines stack-level and frame-level operations. The stack-level operation $\mathrm{activate}(K, \ell)$ changes which frame is active in the stack configuration K. If K does not contain a frame labelled ℓ, the rule ACTIVATE-NEW appends a fresh frame $(\ell, \emptyset, \emptyset)$ to K. If such a frame already exists, the rule ACTIVATE-EXISTING removes it from its current position and re-appends it to the top of the stack. In both cases, the resulting top frame is the frame labelled ℓ.

The remaining rules in Fig. 4 lift the byte-level semantics to operations over a single stack frame. Each frame operation takes a frame $F = (\ell, B, \Sigma)$ and produces an updated frame while preserving the frame label ℓ. The operation $\mathrm{write}(F, o, w, bt)$ is defined when the target interval $[o, o + w)$ lies inside the modelled byte array. It applies the byte-array update $B[o, w \leftarrow bt]$ and preserves the buffer metadata Σ.

The operation $\mathrm{extend}(F, n)$ increases the size of the frame by appending n fresh bytes in the Free state. The operation $\mathrm{push}(F, n, bt)$ appends n initialised bytes to the frame: RWrite appends bytes in the Critical state, whereas nRWrite appends bytes in the Occupied state. The operation $\mathrm{pop}(F, n)$ removes the last n bytes from the frame. The operation $\mathrm{addbuf}(F, \sigma)$ records a stack-buffer descriptor in the frame metadata without changing the byte array. Finally, $\mathrm{setcanary}(F)$ records the presence of a canary on the frame.

These frame operations define local updates to a single frame. Their effect on the complete stack configuration is defined by the action semantics in Fig. 5. The semantic function

$$[\![a]\!] : \mathsf{Stack} \to \mathcal{P}(\mathsf{Stack})$$

maps an abstract memory action a and a stack configuration K to the set of possible successor stack configurations. For direct actions, the operation is applied to the active frame $F_n = \mathrm{top}(K)$, while the caller

Stack operation. $\boxed{Stack_{op} : K \to K'}$

$$\frac{\ell \notin \mathrm{Labels}(K)}{\mathrm{activate}(K, \ell) \to K \cdot (\ell, \emptyset, \emptyset, \mathsf{false})}\ \text{ACTIVATE-NEW}$$

$$\frac{K = K_1 \cdot F \cdot K_2 \qquad F = (\ell, B, \Sigma, \gamma) \qquad \ell \in \mathrm{Labels}(K)}{\mathrm{activate}(K, \ell) \to K_1 \cdot K_2 \cdot F}\ \text{ACTIVATE-EXISTING}$$

Frame operation. $\boxed{Frame_{op} : F \to F'}$

$$\frac{F = (\ell, B, \Sigma, \gamma) \qquad |B| = m \qquad 0 \le o \qquad o + w \le m \qquad B' = B[o, w \leftarrow bt]}{\mathrm{write}(F, o, w, bt) \to (\ell, B', \Sigma, \gamma)}\ \text{WRITE}$$

$$\frac{n > 0 \qquad F = (\ell, B, \Sigma, \gamma)}{\mathrm{extend}(F, n) \to (\ell, B \cdot [\underbrace{\text{Free}, \ldots, \text{Free}}_{n}], \Sigma, \gamma)}\ \text{EXTEND}$$

$$\frac{n > 0 \qquad F = (\ell, B, \Sigma, \gamma) \qquad (bt = \text{RWrite} \wedge bs = \text{Critical}) \vee (bt = \text{nRWrite} \wedge bs = \text{Occupied})}{\mathrm{push}(F, n, bt) \to (\ell, B \cdot [\underbrace{bs, \ldots, bs}_{n}], \Sigma, \gamma)}\ \text{PUSH}$$

$$\frac{n > 0 \qquad F = (\ell, B \cdot C, \Sigma, \gamma) \qquad |C| = n}{\mathrm{pop}(F, n) \to (\ell, B, \Sigma, \gamma)}\ \text{POP}$$

$$\frac{F = (\ell, B, \Sigma, \gamma) \qquad \Sigma : \{\sigma_0, \ldots, \sigma_n\}}{\mathrm{addbuf}(F, \sigma) \to (\ell, B, \{\sigma_0, \ldots, \sigma_n, \sigma\}, \gamma)}\ \text{ADDBUF}$$

$$\frac{F = (\ell, B, \Sigma, \gamma) \qquad \gamma = false}{\mathrm{setcanary}(F) \to (\ell, B, \Sigma, true)}\ \text{SET-CANARY}$$

Fig. 4. Stack and Frame operation semantics.

frames $F_0, \ldots, F_{n-1}$ are preserved. Therefore, Write, Extend, Push, Pop, Canary and Buf produce successor stacks of the form $F_0 \cdots F'_n$, where F'_n is obtained by the corresponding frame operation in Fig. 4.

External calls and loops are handled through summary transitions. $\mathrm{ExtCallSummary}(c, \beta, K)$ returns the stack configurations computed by the emulator for call c starting from stack K, while $\mathrm{LoopSummary}(L, \beta, K)$ returns the stack configurations computed for loop L starting from K, with the symbolic execution bounds $\beta = (i, \tau, \eta)$, where i bounds the number of loop iterations, τ bounds the number of symbolic execution steps, and η bounds the number of active symbolic states. Unlike direct frame operations, these summary transitions are defined over complete stack configurations because the emulated instructions may affect more than the active frame, we will discuss these further in Section 4.2. The action ϵ represents an instruction with no relevant stack-memory effect and therefore leaves the stack unchanged.

Finally, the rule STEP in Fig. 5 combines stack updates with bCFG traversal. From a memory state (ρ, K), the model checker selects an outgoing bCFG edge $(\rho, \rho') \in E$, applies the abstract action $\alpha(\rho)$, and creates a successor state (ρ', K') for every $K' \in [\![\alpha(\rho)]\!](K)$. Thus, the program location evolves according to the bCFG, while the stack configuration evolves according to the semantics of the memory action at the current location.

$$\boxed{[\![a]\!] : \mathsf{Stack} \to \mathcal{P}(\mathsf{Stack})} \qquad \frac{(\rho,\rho') \in E \qquad K' \in [\![\alpha(\rho)]\!](K)}{(\rho, K) \longrightarrow (\rho', K')}\ \textsc{Step}$$

$$[\![\mathsf{Write}(o,w,bt)]\!](F_0 \cdots F_n) \triangleq \{\, F_0 \cdots F'_n \mid \mathrm{write}(F_n, o, w, bt) \to F'_n \,\}$$
$$[\![\mathsf{Extend}(w)]\!](F_0 \cdots F_n) \triangleq \{\, F_0 \cdots F'_n \mid \mathrm{extend}(F_n, w) \to F'_n \,\}$$
$$[\![\mathsf{Push}(w,bt)]\!](F_0 \cdots F_n) \triangleq \{\, F_0 \cdots F'_n \mid \mathrm{push}(F_n, w, bt) \to F'_n \,\}$$
$$[\![\mathsf{Pop}(w)]\!](F_0 \cdots F_n) \triangleq \{\, F_0 \cdots F'_n \mid \mathrm{pop}(F_n, w) \to F'_n \,\}$$
$$[\![\mathsf{Buf}(\sigma)]\!](F_0 \cdots F_n) \triangleq \{\, F_0 \cdots F'_n \mid \mathrm{addbuf}(F_n, \sigma) \to F'_n \,\}$$
$$[\![\mathsf{ActivateFrame}(\ell)]\!](K) \triangleq \{\, K' \mid \mathrm{activate}(K, \ell) \to K' \,\}$$
$$[\![\mathsf{Canary}]\!](F_0 \cdots F_n) \triangleq \{\, F_0 \cdots F'_n \mid \mathrm{setcanary}(F_n) \to F'_n \,\}$$
$$[\![\mathsf{ExtCall}(c,\beta)]\!](K) \triangleq \{\, K' \mid (K, K') \in \mathrm{ExtCallSummary}(c, \beta, K) \,\}$$
$$[\![\mathsf{Loop}(L,\beta)]\!](K) \triangleq \{\, K' \mid (K, K') \in \mathrm{LoopSummary}(L, \beta, K) \,\}$$
$$[\![\epsilon]\!](K) \triangleq \{\, K \,\}.$$

Fig. 5. Action semantics and MemStaCe transition relation.

Memory State Space. A MemStaCe state records both a program location and an abstract stack:

$s = (\rho, K)$.

The memory state space of program P is the transition system

$\mathtt{MemStaCe}_P = (S, s_0, \longrightarrow)$,

where

$S \subseteq V \times \mathsf{Stack}, \qquad s_0 = (\rho_0, K_0)$,

and K_0 is the initial abstract stack. The transition relation $\to \subseteq S \times S$ is defined by the Step rule in Fig. 5: a transition $(\rho, K) \to (\rho', K')$ exists whenever $(\rho, \rho') \in E$ and $K' \in [\![\alpha(\rho)]\!](K)$.

4.2. Summary transitions for external calls and loops

The direct transition rules described above model stack effects that can be derived from a single assembly instruction, such as stack writes, pushes, pops, and frame extensions. However, not all stack effects are visible from a single instruction. Calls to external functions and loop bodies may modify the stack through a sequence of instructions whose effects depend on calling-convention arguments, path conditions, and execution bounds. BASICS represents these through *summary transitions*.

A summary transition is a relation over complete stack configurations. Given a pre-summary stack configuration K, an external-call summary computes the possible successor stack configurations after the summarised call, while a loop summary computes the possible successor stack configurations after a bounded number of loop iterations. These summaries are defined as

$\mathrm{ExtCallSummary}(c, \beta, K) \subseteq \mathsf{Stack}$

$\mathrm{LoopSummary}(L, \beta, K) \subseteq \mathsf{Stack}$,

where $c = \rho$ identifies the program location of the external call site in the recovered bCFG, and $L = \rho$ identifies the program location of the loop header or entry point being summarised. The parameter $\beta = (i, \tau, \eta)$ defines the symbolic-execution bounds used during summary construction, where i bounds the number of loop iterations, τ bounds the number of symbolic execution steps used by angr, and η bounds the number of active symbolic states maintained during the analysis. Although the iteration bound i is only relevant to loop summaries, BASICS uses the same parameterisation $\beta = (i, \tau, \eta)$ for both call and loop summaries to maintain a uniform summary interface. The stack configuration K is the current abstract stack at the program point where a call or loop summary is applied. Therefore, summaries are local to the current state-space construction path; they are not computed from a global initial stack.

User functions. BASICS distinguishes user-functions whose code is present in the target binary from external library calls. These functions are not treated as black boxes, since they appear as assembly code in the recovered bCFG. When the model checker reaches a call to a function whose implementation is present in the recovered bCFG, BASICS creates or reuses the corresponding abstract stack frame and continues traversing the callee's assembly instructions. The stack effects of such functions are then derived by applying the same MemStaCe transition rules used for the rest of the binary.

External C library calls. For calls whose implementation is not traversed as part of the target binary, BASICS uses external-call summaries. When the model checker encounters a call instruction, BASICS first determines whether the target is a recovered user-function or an external function. For the latter, BASICS identifies the function name from the bCFG and consults its C library function database to recover the expected argument types. It then performs a local reverse-flow analysis within the basic block containing the call instruction to recover the values assigned to the System V argument registers, such as `RDI`, `RSI`, `RDX`, `RCX`, `R8`, and `R9`. For instance, in a call to `strcpy`, the destination pointer is expected in `RDI` and the source pointer in `RSI`. If a recovered argument points to a buffer in the current stack frame, BASICS records the corresponding buffer offset and size. BASICS first attempts to summarise supported unsafe C library calls statically. If the destination argument and the relevant stack-buffer metadata can be recovered, the model analysis computes the interval of stack bytes that may be written by the call. This interval is then propagated to the active frame using the byte transition semantics defined earlier. For example, a recovered destination buffer for `strcpy` allows BASICS to compute which bytes of the current frame may be affected by the copy. For memcpy and `memmove`, the length argument is used when it can be recovered; otherwise, the summary must fall back to symbolic execution.

Bounded Symbolic Execution. If static recovery is insufficient, BASICS falls back to the bounded angr-based symbolic execution. The bounded executer searches for a program state that reaches the target call site from the selected analysis entry. This search is bounded by the step limit τ and active-state limit η. When a state reaches the target, BASICS records the relevant stack bytes immediately before the call, executes the call, records the stack bytes again, and compares the contents of both stacks. The byte positions that differ are converted into MemStaCe by the updates and applied to the current abstract stack frame. These states that reach their respective target are cached by the analysis, which records both start and target addresses. This avoids repeating the same exploration whenever multiple summaries require the same target state. However, this execution-based summary is still bounded. If the target address cannot be reached within the configured limits, if symbolic execution produces only unusable states, or if the post-call stack cannot be inspected, the summary is marked as failed.

Concolic Inputs. When a summarised function consumes data from `stdin` or `argv`, BASICS records an input associated with the summary.

In the bounded symbolic execution path, the input is obtained by concretising symbolic bytes: BASICS queries the solver for a satisfying assignment under the path constraints that lead to the summarised call or stack write, and saves the resulting concrete byte string. When the static-summary path is used, no symbolic execution state is available, so BASICS synthesises a stress-test input instead. These inputs are later used during patch validation as a candidate malicious, boundary, or stress-test inputs, but they are not claimed to be complete exploit witnesses for all executions.

Loop summaries. Loops are also represented through bounded summary transitions. BASICS first identifies loop headers and exits from the recovered bCFG. For a loop L, the bounded symbolic executor attempts to reach the loop entry. Once the loop entry is reached, BASICS records the stack contents, executes the loop until either a loop exit is reached or the iteration bound i is exhausted, and then records the stack contents again. The differences between the pre-loop and post-loop contents are translated into byte updates in the corresponding `MemStaCe` frame. If a loop requires more than i iterations to expose a buffer overflow, or if the loop entry or exit cannot be reached within the configured angr bounds, BASICS may fail to compute a complete loop summary. In that case, the analysis continues with the current abstract stack state K, but does not claim completeness for stack effects hidden behind the failed loop summary.

5. Model checking and vulnerability detection

This section describes how BASICS checks stack-security properties over the `MemStaCe` constructed in Section 4. First, it defines the query functions used to observe `MemStaCe` states and explains how these observations give rise to atomic propositions for LTL. It then presents the predefined LTL properties used to capture stack-buffer-overflow patterns. Finally, it describes how BASICS checks these properties using Büchi automata and how violations are reported as counterexample traces that identify the program locations and memory actions responsible for the unsafe write.

5.1. From MemStaCe states to atomic propositions

BASICS checks LTL properties over atomic propositions interpreted on `MemStaCe` states. The query functions in Fig. 6 are not atomic propositions themselves; instead, they provide the state-dependent observations from which atomic propositions are defined. For a memory state $s = (\rho, K)$, an atomic proposition is a Boolean predicate over the current program location, the current stack configuration, and the transition by which s was reached. Although a state is written as $s = (\rho, K)$, BASICS treats `MemStaCe` states as nodes of the unfolded exploration graph. Thus, two distinct nodes may carry the same pair (ρ, K) if they are reached through different paths. This convention makes $\mathsf{PreviousTrans}$ well-defined for every non-initial state. When specifications quantify over `MemStaCe` objects, the quantification is bounded by the finite sets returned by these query functions. For example, a formula of the form

$$\forall F \in \mathsf{Frames}(K).\ \varphi(F)$$

is interpreted at the current state as a finite conjunction over the frames currently represented in K. Therefore, the underlying temporal reasoning remains finite-state, while the specification language can refer directly to frames, buffers, and byte offsets. Fig. 6 defines the query functions used by BASICS. The function $\mathsf{Frames}(K)$ returns the set of abstract stack frames present in the current state, and $\mathsf{Labels}(K)$ returns their labels. Assuming labels are unique within K, $\mathsf{Stack}(K, \ell)$ returns the frame associated with label ℓ. For a frame $F = (\ell, B, \Sigma, \gamma)$, the function $\mathsf{Byte}(F, i)$ returns the abstract byte state stored at offset i, while $\mathsf{HasCanary}(F)$ returns the canary metadata γ.

The function $\mathsf{Buffers}(F)$ returns the buffer metadata associated with frame F. Accordingly, $\mathsf{BuffStart}$, $\mathsf{BuffEnd}$, and $\mathsf{BuffSize}$ expose the start offset, inclusive end offset, and size of a buffer descriptor. These functions allow formulas to relate byte states to buffer boundaries. Finally, $\mathsf{PreviousTrans}(s)$ returns the abstract memory action by which the current state was reached. This query is useful for properties that depend on the last semantic step, for example, whether the current state was reached by an external-call summary or by a loop summary.

$$
\begin{aligned}
\mathsf{Frames}(K) &\triangleq \{F \mid F = (\ell, B, \Sigma, \gamma) \text{ is present in } K\} \\
\mathsf{Labels}(K) &\triangleq \{\ell \mid \exists B, \Sigma, \gamma.\ (\ell, B, \Sigma, \gamma) \in \mathsf{Frames}(K)\} \\
\mathsf{Stack}(K, \ell) &\triangleq F \quad \text{where } F = (\ell, B, \Sigma, \gamma) \in \mathsf{Frames}(K) \\
\mathsf{Byte}(F, i) &\triangleq B(i) \quad \text{where } F = (\ell, B, \Sigma, \gamma) \\
\mathsf{HasCanary}(F) &\triangleq \gamma \quad \text{where } F = (\ell, B, \Sigma, \gamma) \\
\mathsf{Buffers}(F) &\triangleq \Sigma \quad \text{where } F = (\ell, B, \Sigma, \gamma) \\
\mathsf{BuffStart}(\sigma) &\triangleq o \quad \text{where } \sigma = (o, n) \\
\mathsf{BuffEnd}(\sigma) &\triangleq o + n - 1 \quad \text{where } \sigma = (o, n) \\
\mathsf{BuffSize}(\sigma) &\triangleq n \quad \text{where } \sigma = (o, n) \\
\mathsf{PreviousTrans}(s) &\triangleq \begin{cases} \bot, & \text{if } s = s_0, \\ \alpha(\rho_p), & \text{where } \exists!\, (\rho_p, K_p).\ (\rho_p, K_p) \longrightarrow s. \end{cases}
\end{aligned}
$$

Fig. 6. MemStaCe query functions.

5.2. Security properties

Since BASICS targets stack-based buffer overflows, we define eight LTL security properties, shown in Fig. 7. These properties are divided into three classes: *Control-Data Integrity* protect control-sensitive stack data, namely the saved return address (`RIP`), saved base pointer (`RBP`), and stack canary, when present. They require the corresponding byte ranges in each stack frame to remain `C`, i.e., critical. *Unsafe Function Violations* captures calls to intrinsically unsafe library functions, currently `gets`. *Stack-Buffer-Boundary Violations* reason about writes around recovered stack-buffer boundaries, including off-by-one overwrites, underwrites, overflow-by-one patterns, and underflows caused by summarised calls or loops. Next, we give an overview of the security properties they comprise.

Control-Data Integrity. The *RIP Integrity* property checks that the saved return-address region of each stack frame remains critical throughout the execution. A violation indicates that a write has reached control-flow data and may corrupt the return target of the current function. The *RBP Integrity* property is analogous, but protects the saved base pointer. Although overwriting `RBP` does not always immediately redirect control flow, it corrupts the caller's frame layout and is a common symptom of stack-frame overflow. The *Canary Integrity* property applies only to frames in which BASICS recovers a stack canary; it requires the canary byte range to remain critical, and therefore flags writes that would bypass or corrupt compiler-inserted stack protection. The *No Off-by-One Overflow* property captures a narrow control-data corruption pattern in which a write modifies a byte adjacent to critical stack metadata. This property is useful for detecting small overflows that may not overwrite the full return-address or base-pointer range, but still cross the boundary into protected stack data.

Unsafe Function Violations. The *No Gets Usage* property is a separate unsafe-function property: because `gets` performs unbounded input into a destination buffer, BASICS flags any transition corresponding to a call to `gets`.

Stack-Buffer-Boundary Violations. The remaining properties are related to the boundaries of the recovered stack buffers. *No Stack Underwrite* detects writes near the beginning of a buffer that disturb

Control-Data Integrity

RIP Integrity

$$\square\Big(\forall F \in \mathsf{Frames}(K).\ \bigwedge_{i=0}^{7} \mathsf{Byte}(F,i) = \mathsf{C}\Big)$$

RBP Integrity

$$\square\Big(\forall F \in \mathsf{Frames}(K).\ \bigwedge_{i=8}^{15} \mathsf{Byte}(F,i) = \mathsf{C}\Big)$$

Canary Integrity

$$\square\Big(\forall F \in \mathsf{Frames}(K).\ \mathsf{HasCanary}(F) \Rightarrow \bigwedge_{i=16}^{23} \mathsf{Byte}(F,i) = \mathsf{C}\Big)$$

No Off-by-One Overflow

$$\square\neg\Big(\exists F \in \mathsf{Frames}(K).\ \mathsf{Byte}(F,15) = \mathsf{M} \wedge \mathsf{Byte}(F,14) = \mathsf{C}\Big)$$

Unsafe Function Violations

No `gets` Usage

$$\square\neg\,\mathsf{call}(\texttt{gets})$$

Stack-Buffer-Boundary Violations

No Stack Underwrite

$$\square\neg\Big(\exists F \in \mathsf{Frames}(K).\ \exists\sigma \in \mathsf{Buffers}(F).\ \mathsf{BuffSize}(\sigma) > 31 \wedge \mathsf{Byte}(F, \mathsf{BuffStart}(\sigma){+}1) = \mathsf{M} \wedge \mathsf{Byte}(F, \mathsf{BuffStart}(\sigma){+}2) = \mathsf{O}\Big)$$

No Buffer Overflow by One

$$\square\Big(\mathsf{summary} \Rightarrow \neg\Big(\exists F \in \mathsf{Frames}(K).\ \exists\sigma \in \mathsf{Buffers}(F).\ \mathsf{Byte}(F, \mathsf{BuffEnd}(\sigma)) = \mathsf{O} \wedge \mathsf{Byte}(F, \mathsf{BuffEnd}(\sigma){-}1) = \mathsf{M}\Big)\Big)$$

No Underflow

$$\square\neg\Big(\mathsf{summary}\ \mathcal{R}\ \Diamond\Big(\exists F \in \mathsf{Frames}(K).\ \exists\sigma \in \mathsf{Buffers}(F).\ \mathsf{Byte}(F, \mathsf{BuffStart}(\sigma)) = \mathsf{O} \wedge \mathsf{Byte}(F, \mathsf{BuffStart}(\sigma){+}1) = \mathsf{O} \wedge \mathsf{Byte}(F, \mathsf{BuffStart}(\sigma){+}2) \neq \mathsf{O}\Big)\Big)$$

Auxiliary Definitions and Abbreviations

$$\mathsf{C} \equiv \text{Critical}, \quad \mathsf{M} \equiv \text{Modified}, \quad \mathsf{O} \equiv \text{Occupied},$$

$$\mathsf{call}(f) \triangleq \exists c, \beta.\ \mathsf{PreviousTrans}(s) = \mathsf{ExtCall}(c,\beta) \wedge \mathsf{inst}(c) = \texttt{call}\ f,$$

$$\mathsf{summary} \triangleq (\exists c, \beta.\ \mathsf{PrevTrans}(s) = \mathsf{ExtCall}(c,\beta)) \vee (\exists L, \beta.\ \mathsf{PrevTrans}(s) = \mathsf{Loop}(L,\beta)).$$

Fig. 7. BASICS LTL security properties. Auxiliary definitions and abbreviations used by the formulas are given in the bottom panel.

the expected byte-state pattern at the lower boundary. The *No Buffer Overflow by One* property checks whether a loop or call summary has modified bytes immediately adjacent to the end of a recovered buffer, capturing one-byte overwrite patterns. Finally, *No Underflow* detects summary-induced writes that move before the start of a recovered buffer.

5.3. Detecting vulnerabilities

Property Checking. The transition system $\texttt{MemStaCe}_P = (S, s_0, \rightarrow)$ defines the memory states explored by the model checker. To check an LTL property φ, BASICS derives a property-specific Kripke structure

$$\mathcal{K}_{P,\varphi} = (S, s_0, R, AP_\varphi, \mathcal{L}_\varphi)$$

from $\texttt{MemStaCe}_P$. This construction keeps the same states and initial state, adds stuttering self-loops to terminal states, and labels each state with the atomic propositions of φ that hold there. The set AP_φ contains the atomic propositions that occur in φ. Each proposition is a Boolean predicate over MemStaCe states, evaluated using the query functions in Fig. 6. Thus,

$$\mathcal{L}_\varphi(s) = \{\, p \in AP_\varphi \mid [\![p]\!]_s = \mathsf{true} \,\}.$$

The query functions provide the observation interface over MemStaCe states, while AP_φ contains the Boolean propositions used by the LTL checker. The Kripke transition relation R is the totalised version of the MemStaCe transition relation:

$$(s,s') \in R \quad \text{iff} \quad s \longrightarrow s' \ \vee\ \big(s = s' \wedge \nexists t \in S.\ s \longrightarrow t\big).$$

The second case adds a stuttering self-loop to terminal states, allowing finite MemStaCe executions to be interpreted as infinite paths, as required by LTL, without adding any memory effect. Each property φ is translated into a Büchi automaton, and BASICS checks $\mathcal{K}_{P,\varphi}$ against the automaton for $\neg\varphi$. A reachable accepting run corresponds to an execution of the program abstraction that violates φ. Since the predefined properties are safety-oriented invariants, violations are witnessed by finite bad prefixes. In the implementation, BASICS does not explicitly construct the full product graph. Instead, it traverses $\texttt{MemStaCe}_P$ while maintaining the current automaton state for each property. At each memory state, the automaton guards are evaluated through $\mathcal{L}_\varphi$, and the automaton state is advanced accordingly.

Counterexample Traces. A vulnerability candidate is reported when the automaton for a security property reaches a violating configuration along a reachable MemStaCe path. The report contains the violated property and a counterexample trace that explains how the violating state was reached. A trace is a sequence of tuples

$$\langle addr, instr, MemAct \rangle,$$

where $addr$ is the instruction address, $instr$ is the disassembled instruction, and $MemAct$ is the static memory action or summary action applied at that step. The trace links the LTL violation back to concrete binary locations and to the stack-memory updates that caused the violation.

Example. To illustrate the creation of the MemStaCe and the detection of a security property violation, Fig. 8 shows the step-by-step construction of the MemStaCe for the `copy` function (Listing 2). The process begins by initialising the stack frame with 8 bytes in the *Critical* state, and then pushing the stack base pointer onto the stack frame ❶. The frame is then extended ❷; newly allocated bytes are marked *Free*, reflecting the uninitialised space. Later, instruction ❸ prepares the call to `strcpy`, and although instruction ❹ does not modify

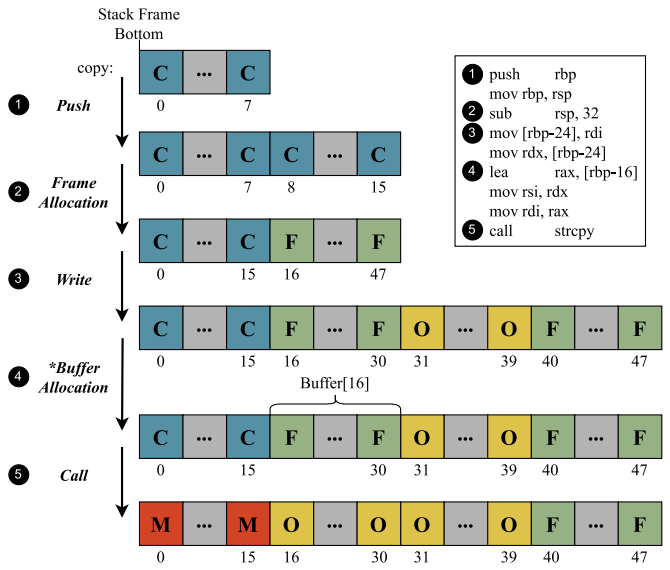

Fig. 8. Example of the MemStaCe for the copy function in Listing 2, where **C**: Critical, **O**: Occupied, **F**: Free, **M**: Modified.

```
<0x401169; endbr64; ActivateFrame("copy")>
<0x40116d; push rbp; Push(8, RWrite)>
<0x401171; sub rsp, 0x20; Extend(32)>
<0x401175; mov dword ptr [rbp-0x14], edi; Write(19, 4,
 ↪ nRWrite)>
<0x401178; mov qword ptr [rbp-0x20], rsi; Write(31, 8,
 ↪ nRWrite)>
<0x401187; lea rax, [rbp-0xa]; Buf((10, 2))>
<0x401191; call 0x401060 <strcpy>; ExtCall(0x401191)>
```

Listing 3: Counter-Example Trace for a program that violates the *RIP Integrity* Security Property.

memory directly, it represents a transition that identifies a target buffer address for later use. The final instruction ❺ performs the actual call to `strcpy`, whose effect is summarised in the MemStaCe. However, due to a buffer overflow, the write extends into adjacent memory and modifies *Critical* bytes, transitioning them to the *Modified* state. This last transition triggers a violation of the RIP Integrity Property, resulting in the trace being flagged by the model checker as a counterexample. Listing 3 shows the counterexample trace for the same violation. The trace contains the frame setup, the local stack allocation, and the call to `strcpy`. The call summary writes beyond the recovered destination buffer and modifies bytes in the saved return-address range. Since *RIP Integrity* requires these bytes to remain critical in all reachable states, BASICS reports the trace as a counterexample.

5.4. Bounded soundness and completeness conditions

The model-checking result is relative to the constructed MemStaCe. BASICS therefore does not claim unconditional soundness or completeness for arbitrary binary executions. Instead, the following conditions characterise the validity of counterexamples relative to the constructed abstraction and the circumstances under which concrete stack-buffer overflows are represented by the bounded analysis.

Condition 1 (*Bounded Soundness Condition*)**.** *Let π be a counterexample trace reported by* BASICS *for property φ. If the recovered bCFG contains the corresponding program path, the instruction-to-action translation is correct for every instruction in π, and all call and loop summaries used in π accurately represent the stack bytes written by the corresponding bounded execution segments, then π witnesses a violation of φ in the bounded MemStaCe abstraction.*

Argument. By construction, each transition in π follows an edge of the recovered bCFG and applies the action semantics defined in Section 4. Under the stated assumptions, these actions preserve the relevant stack-byte effects of the corresponding binary instructions or summaries. The labelling function exposes these byte states and state predicates to the property automaton. Therefore, if the automaton reaches a violating configuration along π, the corresponding sequence of MemStaCe states violates φ.

Condition 2 (*Bounded Completeness Condition*)**.** *Assume a concrete stack-buffer overflow occurs along an execution path contained in the recovered bCFG, within the configured loop bound and angr execution limits, and through instructions or summarised calls whose stack effects are captured by* BASICS*. If the overflow changes byte states in a way covered by one of the predefined security properties, then the constructed MemStaCe contains a corresponding violating path, which* BASICS *can report as a counterexample.*

Argument. The assumptions ensure that the relevant execution path is represented in the recovered bCFG and that each stack effect along the path is reflected by a direct MemStaCe transition or a successful summary transition. Therefore, the write that causes the overflow appears as a modified byte pattern in some reachable MemStaCe state. Since the predefined property is evaluated on these byte states, the corresponding automaton detects the violation, and the traversal can emit the associated counterexample trace.

These conditions are bounded by design. BASICS may miss vulnerabilities whose paths are absent from the recovered bCFG, whose effects occur beyond the loop iteration bound, whose calls cannot be summarised, or whose stack writes are hidden by unsupported instructions, unresolved indirect calls, or failed argument recovery. Conversely, conservative summaries may introduce false positives when a statically possible overwrite is infeasible in the concrete program. We empirically evaluate these cases in Section 8.

6. Patching vulnerabilities and bounded validation

When the model checker reports a security-property violation, BASICS attempts to repair the binary if the counterexample can be linked to a supported patchable sink. In the case of multiple property violations identifying the same call site, BASICS consolidates them into a single patch sink. On the other hand, distinct supported call sites are processed independently, and the patch template is selected according to the normalised called-function name. The repair pipeline has four stages: identifying the sink responsible for the violation, generating a patch payload for that sink, installing the payload in the binary, and validating the resulting patched binary. The validation step is bounded: it provides evidence that the patch removes the observed vulnerable behaviour and preserves behaviour on tested benign and boundary inputs, but it does not prove whole-program functional equivalence.

Vulnerability Identification. Using the counterexample trace produced for the violated security property, a reverse flow analysis is performed, tracing the execution backwards to identify the instruction behind the violation, as well as its location in bCFG and the user function address that contains such instruction. Since most BOs originate from function calls, the vulnerability is often linked to the `call` instruction at the end of the trace. For example, in the counterexample of Listing 3, this process starts from the end of the counterexample, following a backwards analysis until finding the instruction `call 0x401060`, a function call presented in BO exploitation. For vulnerabilities that are not related to external function calls, such as buffer underflows caused by loops, the same reverse analysis is applied, but looking for the entry point of the loop instead of a function call. Also,

Table 1
Mapping of security properties to CWE classes.

Security property	CWE class
RIP Integrity	CWE-121, CWE-787
RBP Integrity	CWE-121, CWE-787
Canary Integrity	CWE-121, CWE-787
No `gets` Usage	CWE-676
No Off-by-One Overflow	CWE-193, CWE-121, CWE-787
No Buffer Overflow by One	CWE-193, CWE-119, CWE-121, CWE-787
No Stack Underwrite	CWE-124
No Underflow	CWE-124

using the *User Function Map*, the name of the user function is obtained by searching for the user function address extracted from the backwards analysis. Additionally, to support developers and analysts with the types of BOs identified, the Common Weakness Enumeration (CWE) classes are provided for the properties that are violated. For this, the *Vulnerability CWE Class* database, presented in Table 1, contains a map from security properties to one or more CWE classes. For example, the return address integrity property (RIP Integrity in Fig. 7) is associated with CWE-121 and CWE-787 (MITRE Corporation - Common Weakness Enumeration, 2025a,b), representing stack-based buffer overflow and its more general out-of-bounds-write class, respectively.

Patch Template Generation. For each patchable sink, BASICS selects a patch template according to the recovered library function. The template is instantiated using the pre-call state recovered during MemStaCe construction and call-summary generation. This state contains the relevant System V AMD64 argument registers, the destination object, the recovered destination size when available, stack-canary metadata, and the continuation address after the replaced call.

Templates have two variants. If the destination stack-buffer size is recovered statically, the known-size variant receives this bound directly. If the bound cannot be recovered statically, the unknown-size variant computes a conservative runtime bound from the stack-frame layout, using RBP, the destination pointer, and the canary offset when a canary is present. Runtime-bound templates are used only when the destination is identified as a stack object in the current frame; BASICS skips destinations known to be heap-backed or otherwise outside the supported stack-object model. Patch payloads also preserve the expected library-call return convention, so that callers observing the return value see the payload as a replacement for the original call.

All templates follow the same replacement pattern. The inserted payload is called instead of the unsafe library routine, receives the original call arguments, and enforces a bound before writing to the destination object. String templates limit the copy to the destination capacity and explicitly write a trailing NUL; concatenation templates also account for bytes already present in the destination; raw-copy templates clamp the requested length; and input templates bind the token or formatted assignment that writes into the stack object. After the payload finishes, execution resumes at the instruction following the original call. Thus, the patch changes the unsafe write behaviour without changing the surrounding control flow.

Listing 4 shows the statically known-size `strcpy` payload. The payload bounds the copy to `size - 1`, forces a terminating NUL byte, handles degenerate bounds, and writes the expected `strcpy` return value back to the caller-visible return slot. When the destination buffer size cannot be recovered statically, the patch approximates a runtime bound by computing the distance, in bytes, between RBP and RDI. Listing 5 shows this dynamic variant.

Patch Installation. BASICS installs patches by using the trampoline technique (Duck et al., 2020) that allows binary rewriting. For each patchable call-site, the patcher creates a rewriting rule that replaces the unsafe library call with the selected `apply_patch` payload. The rule passes the arguments recovered at the original call site, including the relevant argument registers, the return-value location, and the

```c
#include "stdlib.c"
#include "patch_helpers.h"

void apply_patch(char *rdi, char *rsi, intptr_t *ret,
    intptr_t size)
{
    if (size <= 1) {
        basics_set_return(ret, (intptr_t)rdi);
        return;
    }
    strncpy(rdi, rsi, size - 1);
    rdi[size - 1] = '\0';
    basics_set_return(ret, (intptr_t)rdi);
}
```

Listing 4: Strcpy function patch template with statically determined buffer sizes.

```c
#include "stdlib.c"

void apply_patch(bool canary, void *rbp, char *rdi,
    char *rsi)
{
    long offset = 0;
    if (canary)
    {
        offset = 8;
    }
    long size = (long)rbp - (long)rdi - offset;
    strncpy(rdi, rsi, size - 1);
}
```

Listing 5: Strcpy function patch template with buffer sizes determined at runtime.

destination bound or runtime-bound metadata. At runtime, execution reaches the original call site, is redirected to the compiled payload, performs the bounded operation, and then resumes at the instruction immediately following the replaced call.

Before reporting a patched binary, BASICS checks that all requested rewriting rules were installed. The patcher records the original and patched binary paths, the patched call-sites, continuation addresses, selected templates, size mode, bound metadata, and E9Patch installation result in a patch manifest. This manifest is consumed by the validator and provides an auditable record of which binary locations were rewritten. Because E9Patch can implement redirection through trampoline or loader mappings, the original call may still be visible in ordinary static disassembly; BASICS therefore treats the installation result and the patch manifest as evidence that the patch was successfully installed.

Bounded Validation Evidence. After rewriting, BASICS executes the original and patched binaries on the same validation cases. The case label is assigned before validation, based on where the input came from. If the user provides a validation JSON file, each case uses the file's `kind` field, which must be `malicious`, `benign`, or `boundary`. Otherwise, inputs produced by the concolic execution phase for a violating path are treated as malicious. If no such input is available, BASICS creates a cyclic long-string fallback and treats it as malicious. The validator then adds two preservation inputs: a short benign string and a boundary string whose length is derived from the smallest recovered destination bound.

Malicious cases test remediation: the original binary must expose the reported vulnerable behaviour, and the patched binary must avoid the corresponding crash, timeout, or vulnerable-site failure. Benign and boundary cases test bounded preservation: the patched binary must

preserve the original exit code and standard output under the selected comparison policy, and it must not introduce a crash or timeout. Thus, the labels are not inferred from the patched result; they define what the validator is supposed to check for that input.

The validator also checks whether the test input actually reaches the patched call. It runs the patched binary under runtime observation, translates the recorded patch address to the address used at runtime, records whether execution reaches the patched call site, and captures the main stack-related registers around the patched execution. This distinguishes a test that never reaches the patched code from a test that runs through the patch and continues normally.

The validation report separates remediation from preservation. A remediation pass means that the patched binary no longer shows the vulnerable behaviour observed in the original execution. A preservation pass means that the tested benign and boundary inputs produce the same observable behaviour, specifically the same exit code and standard output, as the original binary. Otherwise, the patch is classified as not behaviour preserving. This includes cases where the patched execution changes observable behaviour, where the validation input does not execute the patched code, or where the original binary already behaves inconsistently across repeated executions of the same input, preventing a reliable before-and-after comparison. These checks are limited to the executed inputs; they provide bounded validation evidence for the installed patch, not a proof of full functional equivalence.

7. Implementation

BASICS (*Binary Analysis and Stack Integrity Checker System*) is implemented in Python 3.11 for x86-64 Linux ELF binaries. It uses angr (Shoshitaishvili et al., 2016) for binary loading, disassembly, control-flow recovery, and bounded symbolic execution; Spot (Duret-Lutz et al., 2022) for LTL-to-Büchi translation; E9Patch (Duck et al., 2020) for binary rewriting; and GNU Debugger (GDB) (GNU and Fee Software Foundation, 1986) for runtime patch validation. The implementation follows the four modules in Fig. 1: the Binary Data Extractor, the Security Property Converter, the Model Checker, and the Vulnerability Patcher and Validator. BASICS is an open source tool, and it is available at Ferreirinha and Medeiros (2026).

Binary Data Extractor. The extractor loads the target binary with angr using `auto_load_libs=False`. BASICS does not analyse external libraries as part of the target program; instead, supported library calls are modelled by the summary mechanisms described below. The extractor records binary metadata, including the file path, file size, loader ranges, entry point, sections, segments, and recovered functions. It then constructs the recovered binary control-flow graph (bCFG) used by the rest of the pipeline.

BASICS supports two analysis modes. In whole-program mode, analysis starts from the selected program entry point and explores the bCFG from that entry. This mode preserves more calling context, but it can become expensive for large binaries. In function-analysis mode, BASICS recovers the bCFG once, identifies user functions with stack frames, and schedules each such function as an independent analysis entry. Each function entry is checked using the same MemStaCe construction, summary generation, and LTL model-checking code as whole-program analysis. This mode allows BASICS to scale to large binary collections by analysing functions separately, but its reports are treated as candidate findings because caller context may be incomplete. Candidate reports from function-analysis mode can then be rechecked from a containing caller or program entry when confirmation is required.

Security Property Converter. Security properties are written as LTL formulas over the MemStaCe observations defined in Section 5. The converter maps these formulas to BASICS predicates, uses Spot to generate Büchi automata, and stores the automata in the format used by the model checker. Generated automata are cached across runs.

Model Checker. The model checker constructs the `MemStaCe` by traversing the recovered bCFG and applying the stack-memory semantics from Section 4. Each state records the current program location, stack frames, byte states, buffer metadata, canary metadata, and the current abstract memory action. Previously processed combinations of program location and abstract stack state are reused to avoid repeatedly expanding the same region.

Calls and loops are summarised before their effects are added to the `MemStaCe`. For supported C library routines, BASICS first recovers the call arguments, stack destination, destination size, and affected byte range directly at the call site. When local recovery is not enough, BASICS uses bounded symbolic execution to summarise the call or loop: it searches from the current analysis entry to the target, compares stack bytes before and after the operation, and converts the differences into `MemStaCe` updates. If the target is not reached within the configured bounds, BASICS records the unresolved summary and continues from the current abstract stack state.

After building the reachable `MemStaCe`, the model checker evaluates the Büchi automata generated from the LTL properties. If a property is violated, BASICS reports the violated property, the counterexample trace, and any concrete inputs generated during summary construction. Otherwise, it reports the properties that hold over the explored `MemStaCe`.

Vulnerability Patcher and Validator. The patcher uses counterexample traces to identify supported vulnerable library calls and select matching patch templates. The current patch set covers `strcpy`, `gets`, `scanf`, `fscanf`, `sscanf`, `strcat`, `sprintf`, `memcpy`, and `memmove`. Templates either use a recovered destination size or compute a runtime bound from the stack frame.

Larger routines shared by multiple templates are factored into `patch_helpers.h`. These helpers implement bounded token reads, bounded string extraction, the supported `scanf/fscanf/sscanf` conversions, bounded `gets`-style line reading, and uniform return-value handling. Keeping this logic in a helper header keeps individual templates small while preserving the same guarantee: helpers write at most the recovered capacity, terminate string destinations when applicable, and return a value compatible with the replaced library routine.

BASICS installs selected patches with E9Patch (Duck et al., 2020). For each patched call site, it records the selected template, recovered or runtime bound, continuation address, and E9Patch installation result in a patch manifest. A binary is reported as patched only when all requested call sites are successfully rewritten.

After patch installation, the validator runs the original and patched binaries on the same malicious, benign, and boundary inputs. Malicious inputs check whether the patch removes the observed crash or vulnerable behaviour. Benign and boundary inputs check that the patched binary preserves the original exit code and standard output and does not introduce a crash or timeout.

During this runtime validation stage, BASICS also uses GDB to observe the patched execution. For each validation input, GDB starts the patched binary, resolves the runtime address of the patched call site, sets a breakpoint at that site, and records whether execution reaches the patch. When the breakpoint is hit, BASICS records the main stack-related registers around the patched call. The final report separates vulnerability remediation from bounded functional preservation. A patch is marked as remediated when the malicious input no longer triggers the observed crash or property violation. It is marked as behaviour preserving only when the patched binary keeps the original exit code and standard output on benign and boundary inputs without introducing a crash or timeout. Otherwise, the patch is classified as not behaviour preserving, including cases where the validation input does not reach the patched call site, the original execution does not provide a stable baseline, or the patched execution produces different observable behaviour.

8. Experimental evaluation

We evaluate BASICS along five dimensions: the fidelity of the generated `MemStaCe`, the effectiveness of the LTL security properties as vulnerability specifications, the ability of the model checker to detect BOs, the effectiveness of generated patches, and the scalability of the implementation on real binaries. This leads to the following research questions: **Q1.** Does BASICS detect stack-based buffer overflows with high precision? **Q2.** How does BASICS compare with existing analysis tools on the same datasets? **Q3.** Are generated patches effective on patchable vulnerabilities? **Q4.** Can BASICS be applied to real open-source programs and Linux package binaries? **Q5.** What are the main causes of false positives (FPs) and false negatives (FNs)?

8.1. Evaluation setup and grading protocol

8.1.1. Evaluation setup

The evaluation is organised into four parts. First, we measure detection accuracy and execution performance on controlled benchmark datasets with known vulnerable and non-vulnerable cases. Second, we evaluate binary patch generation and bounded patch validation on patchable programs from one of the controlled datasets. Third, we run BASICS on larger open-source C/C++ projects to assess scalability under realistic program structure. Finally, we perform a Linux package sweep to evaluate whether the function-by-function analysis mode can be scheduled across many independent real-world binaries.

For controlled benchmarks, we used the Juliet C/C++ test suite (NIST, 2017) and NIST SARD (NIST, 2021). The Juliet subset contains 3524 isolated CWE-121 binaries, with 1762 vulnerable and 1762 non-vulnerable cases. The SARD subset contains 151 C programs, with 51 vulnerable and 100 non-vulnerable cases. For open-source projects and Linux packages, we used 5 (3 and 2, respectively) that contain 292 binaries. For Linux packages, we specifically chose the Core Linux package and the Additional packages associated with it, which together comprise 58 packages.

All experiments were run on Ubuntu 24.04.4 LTS on a machine with 48 AMD EPYC 7643 hardware threads and 251 GiB of RAM. The benchmark harness schedules independent cases in parallel and records per-case status, elapsed time, reported vulnerabilities, and confusion-matrix data. Unless stated otherwise, the reported time is the sum of per-case elapsed times; for the large sweeps, we also report wall-clock time measured from the first case start to the last case finish.

8.1.2. Grading protocol

A binary is classified as vulnerable by BASICS if the model checker reports at least one violation of the stack-security properties defined in Fig. 7. Since for Juliet and SARD, the benchmark labels provide ground truth, each report can be classified as a true positive or false positive at the benchmark-case level. For open-source and Linux package sweeps, since they do not provide such labels, reports from these sweeps are therefore treated as candidate vulnerabilities until they are confirmed by follow-up manual review.

Additionally, using Juliet and SARD, we compare BASICS with four reproducible tools that can be applied directly to automated vulnerability analysis, since other tools discussed later in Section 11 either do not provide publicly available implementations or require manually prepared targets and inputs. Hence, for BASICS comparison, we chose the tools CWE_Checker (Barabosch and Enkelmann, 2024), Flawfinder (Wheeler, 2025), Manticore (Mossberg et al., 2019), and BinAbsInspector (Keenlab, 2022). These baselines are not all dedicated binary-level stack-buffer-overflow classifiers: Flawfinder is source-code only, Manticore is a general symbolic-execution engine, and CWE_Checker and BinAbsInspector combine memory-safety checks with broader dangerous-function diagnostics. For tools that provide both direct memory-safety reports and unsafe-function warnings, we include both classes in the comparison. Consequently, CWE_Checker and BinAbsInspector are evaluated with CWE-676 dangerous-function checks enabled in addition to stack memory-safety reports. Since CWE-676 represents unsafe API usage rather than direct evidence of stack corruption, this configuration can increase FPs.

In this comparison, positive reports on vulnerable benchmark-cases are true positives, while positive reports on non-vulnerable benchmarks are FPs. Specifically, BASICS is graded positive when the model checker reports a violation of the stack-security LTL properties. CWE_Checker is graded positive when its JSON output contains CWE-119, CWE-787, or CWE-676. Flawfinder is graded positive when its CSV output contains a buffer-overflow CWE-family report, excluding the generic static `char` array rule. Manticore is graded positive only when generated test cases trigger a crash, segmentation fault, or invalid-memory access. BinAbsInspector is graded positive when it reports CWE-676, stack-relevant CWE-119/CWE-787, or stack out-of-bounds violations; heap out-of-bounds reports are filtered because the benchmark targets stack vulnerabilities.

For tools with failed benchmark cases, metrics are computed only over completed cases; *Errors* are reported in cases of timed out, failed to build or could not be run, and therefore excluded from the confusion matrix.

8.2. Detection on SARD and Juliet subsets

8.2.1. SARD

Table 2 shows the SARD results. BASICS completed all 151 cases and achieved the best accuracy and F1-score among the five tools. Its precision was 86.4%, with 6 FPs and 13 FNs. Flawfinder reached a higher recall, but at the cost of a large number of FPs. CWE_Checker detected the largest number of vulnerable cases among the binary tools, but most of its positive signals came from CWE-676 dangerous-function warnings rather than direct stack-corruption reports. This improves recall, but also produces 70 FPs because unsafe API use is only a candidate vulnerability signal. BinAbsInspector produced no positives on SARD, even with CWE-676 enabled, giving perfect specificity but zero recall. Manticore produced useful results on the cases it completed, but timed out on 27 out of the 151 cases.

BASICS also completed the benchmark substantially faster than the binary-analysis baselines. Its aggregated analysis time was 13 m 0s, compared to approximately one hour for CWE Checker, BinAbsInspector, and Manticore. Flawfinder completed faster still, but operates directly on source code rather than binary state exploration.

These results answer Q1. positively on small and medium-sized benchmark programs: when BASICS constructs a sufficiently precise state space, property violations usually correspond to real stack corruption. The remaining FNs are mainly caused by imprecise modelling of library functions and by state-space truncation in loops. The FPs occur when conservative stack-state approximations mark frame metadata as modified even though the source program uses bounded operations.

8.2.2. Juliet

Table 3 summarises the Juliet CWE-121 results. The Juliet dataset is substantially harder for BASICS because many cases contain deeper call chains, loop-dependent writes, and library functions whose effects must be approximated. BASICS still preserves high precision, 86.1%, but its recall drops to 51.1%. This indicates that reported violations are usually meaningful, while missed cases are primarily due to incomplete state-space construction or imprecise function modelling rather than to weak security properties.

CWE_Checker detected the largest number of vulnerable cases among the binary-analysis tools, but this came with a high FP rate because many safe Juliet binaries still call APIs classified as dangerous. BinAbsInspector had the opposite behaviour: it produced few reports, so its FP rate was low, but it missed nearly all vulnerable cases. Flawfinder finds more vulnerable cases than BASICS but also reports many non-vulnerable programs as vulnerable, i.e., FPs. Manticore

Table 2
Detection results on the SARD dataset.

Tool	Errors	TP	FP	TN	FN	Precision	Recall	F1	Accuracy	Sum time
BASICS	0	38	6	94	13	86.4%	74.5%	80.0%	87.4%	13 m 0 s
BinAbsInspector	0	0	0	100	51	N/A	0.0%	0.0%	66.2%	1 h 2 m 7 s
CWE_Checker	0	42	70	30	9	37.5%	82.4%	51.5%	47.7%	59 m 9 s
Flawfinder	0	48	93	7	3	34.0%	94.1%	50.0%	36.4%	11.8 s
Manticore	27	16	9	75	24	64.0%	40.0%	49.2%	73.4%	1 h 0 m 9 s

Table 3
Detection results on the Juliet CWE-121 dataset.

Tool	Errors	TP	FP	TN	FN	Precision	Recall	F1	Accuracy	Sum time
BASICS	0	900	145	1617	862	86.1%	51.1%	64.1%	71.4%	14 h 40 m 41 s
BinAbsInspector	0	8	36	1726	1754	18.2%	0.5%	0.9%	49.2%	29 h 39 m 37 s
CWE_Checker	0	1472	1454	308	290	50.3%	83.5%	62.8%	50.5%	32 h 39 m 21 s
Flawfinder	0	1108	1090	672	654	50.4%	62.9%	56.0%	50.5%	5 m 38 s
Manticore	101	671	0	1712	1040	100.0%	39.2%	56.3%	69.6%	29 h 23 m 5 s

Table 4
Case-level detection overlap between BASICS and CWE_Checker.

Dataset	Both	BASICS only	CWE_Checker only	Neither
SARD	30	8	12	1
Juliet	794	106	678	184

achieved high precision on the completed subset, but timed out on 101 cases.

Runtime differences were more pronounced on Juliet. BASICS completed the full benchmark in 14 h 40m, while CWE Checker, BinAbsInspector, and Manticore each required approximately 29–33 h of aggregated analysis time. Flawfinder was substantially faster because it performs lightweight source-level pattern matching rather than binary-level state exploration and verification.

We further categorised the 862 false negatives produced by BASICS on Juliet according to their dominant cause. Approximately 44% (379 cases) resulted from incomplete modelling of copy, formatting, inlined-write, or wide-character operations. Approximately 37% (319 cases) arose from missing execution context or bounded exploration, including runtime-dependent inputs, and loops. The remaining 19% (164 cases) were source-level off-by-one cases where the out-of-bounds write fell into compiler-added padding or alignment space. As a result, the write did not overwrite an adjacent stack object in the compiled binary.

8.2.3. Results discussion on both subsets

Since BASICS and CWE_Checker were the two best tools on both subsets, a further analysis was done to understand where they overlap or do not overlap in vulnerable case detection. Table 4 compares both tools at the case level, considering only ground-truth vulnerable cases. On SARD, both tools detected the same 30 cases, while 8 were detected only by BASICS and 12 only by CWE_Checker. On Juliet, both tools detected 794 of the same cases, while 106 were unique to BASICS and 678 to CWE_Checker. Although most of BASICS's true positives overlap with those of CWE_Checker, BASICS still identifies vulnerable cases that CWE_Checker misses.

Together, the SARD and Juliet results answer Q2. BASICS provides the strongest overall balance between precision, recall, accuracy, and analysis cost among the evaluated tools. The comparison also highlights important differences between analysis approaches: source-level pattern matching, binary dangerous-function diagnostics, symbolic test generation, and LTL-based binary model checking expose different forms of evidence and therefore produce different precision–recall tradeoffs. BASICS's main limitation is recall on complex Juliet programs, which is consistent with the cost of path exploration, loop handling, and function emulation in binary-level symbolic execution.

Table 5
SARD benchmark composition and structured patch validation.

Class	Cases	Vuln.	Patched	Passed	Failed preservation
`gets`	2	2	2	2	0
`fscanf`	7	4	4	4	0
`scanf`	16	6	6	6	0
`sscanf`	3	1	2	2	0
`strcpy`	51	19	15	14	1
`strcpy` underwrite	5	5	5	5	0
`sprintf`	18	13	12	8	4
`vsprintf`	4	1	0	0	0
Other not-vulnerable	45	0	0	0	0
Total	**151**	**51**	**46**	**41**	**5**

8.3. Patching on SARD

We also evaluated patch generation and bounded patch validation on SARD. Each patch records a manifest and is validated over generated or provided malicious, benign, and boundary inputs. The Validator reports vulnerability-mitigation evidence separately from bounded functional-preservation evidence, and does not claim full functional preservation for the patching process.

Table 5 shows both the functions and patterns represented in the SARD subset and the patch-validation outcome. The dataset contains 151 cases, of which 51 contain BO vulnerabilities. The Patcher produced 46 patched binaries. 41 patched binaries passed both vulnerability mitigation and bounded functional-preservation checks. The remaining 5 patched binaries removed the observed vulnerable behaviour but failed bounded preservation: the patched executions differed from the original executions on the validator's benign or boundary inputs.

The input-function classes were the most direct cases. Both `gets` cases, all four patched `fscanf` cases, all six patched `scanf` cases, and both patched `sscanf` cases passed validation. The `strcpy` class produced 15 patched binaries; 14 passed validation and one failed preservation. All 5 `strcpy` underwrite cases passed validation. The `sprintf` class produced 12 patched binaries; 8 passed validation and 4 failed preservation. These failures occurred in format-sensitive executions where the patch removed the unsafe write but changed observable behaviour, such as formatted output or termination behaviour. No patch was produced for the vulnerable `vsprintf` case.

These results answer Q3. positively within the bounds of the validator. The generated patches removed the observed vulnerable behaviour in all patched binaries, and 41 out of the 46 patched binaries preserved the original exit code and standard output on the validator's benign and boundary inputs. The remaining 5 cases changed observable behaviour after patching, primarily in format-sensitive functions such as `sprintf`.

Table 6
Average per-case pipeline-stage runtime from a separate instrumented profiling experiment. SARD uses all 151 cases, while Juliet uses a stratified 200-case subset containing 100 matched bad/good pairs. Values are arithmetic mean ± standard deviation in seconds. End-to-end time includes the measured stages and unattributed runtime overhead.

Dataset	Disassembly/bCFG	MemStaCe	LTL checking	Patch generation	Patch Validation	End-to-end
SARD	0.078 ± 0.005 s	0.052 ± 0.277 s	0.048 ± 0.024 s	0.006 ± 0.009 s	1.028 ± 2.027 s	1.338 ± 2.050 s
Juliet	0.244 ± 0.012 s	13.741 ± 42.646 s	0.211 ± 0.146 s	N/A	N/A	14.317 ± 42.657 s

Table 7
Open-source project sweep and function-level candidate filtering. *Filtered (%)* reports the percentage of initial candidates removed, and *Cand. bins* counts binaries containing at least one final candidate.

Project	LoC	Binaries	Errors	Funcs	Initial cand.	Filtered	Final cand.	Filtered (%)	Cand. bins	Sum time	Wall time	Total bin. size
cJSON	22,383	23	0	2537	0	0	0	–	0	7h 16 m 25 s	57 m 21 s	2.2 MiB
libplctag	95,649	25	4	12,440	0	0	0	–	0	5h 12 m 26 s	57 m 0 s	172.3 MiB
SOEM	25,305	7	2	736	7	1	6	14.3%	1	1h 14 m 28 s	30 m 4 s	1.9 MiB
Total	**143,337**	**55**	**6**	**15,713**	**7**	**1**	**6**	**14.3%**	**1**	**13 h 43 m 19 s**	**2 h 24 m 25 s**	**176.4 MiB**

Table 8
Linux package sweep and function-level candidate filtering. *Filtered (%)* reports the percentage of initial candidates removed, and *Cand. bins* counts binaries containing at least one final candidate.

Sweep	Packages	Binaries	Funcs	Initial cand.	Filtered	Final cand.	Filtered (%)	Cand. bins	Sum time	Wall time	Total bin. size
Core Linux packages	21	117	1737	38	38	0	100.0%	0	14 h 10 m 54 s	1 h 50 m 3 s	32.5 MiB
Additional packages	37	120	4016	221	220	1	99.5%	1	11h 12 m 18 s	1 h 28 m 35 s	14.0 MiB
Combined	**58**	**237**	**5,753**	**259**	**258**	**1**	**99.6%**	**1**	**25h 23 m 12 s**	**3h 18 m 47 s**	**46.6 MiB**

8.4. Pipeline runtime profiling on ground-truth subsets

We conducted a separate profiling experiment to complement the aggregated runtime results reported in Section 8.2, Tables 2 and 3. This experiment instrumented BASICS to measure the elapsed time spent in binary disassembly and bCFG recovery, `MemStaCe` construction, LTL model checking, patch generation, and patch validation. It uses all 151 SARD cases and a stratified 200-case Juliet subset containing 100 matched bad/good pairs. Table 6 reports the average and standard deviation of the per-case elapsed time for each stage. End-to-end time includes the measured stages and unattributed pipeline overhead.

`MemStaCe` construction dominates the Juliet profiling results, accounting for 13.741 s of the 14.317 s average end-to-end runtime. For SARD, patch validation is the largest component, averaging 1.028 s of the 1.338 s end-to-end runtime, while disassembly, LTL checking, and patch generation each remain below 0.1 s on average. Patch generation and validation are not reported for Juliet because the Juliet profiling experiment evaluates detection only.

8.5. Scalability on open-source projects

To evaluate the BASICS's scalability on real programs, we built and scanned three larger open-source C/C++ projects: cJSON,[1] libplctag,[2] and SOEM.[3] Because whole-binary analysis can exhaust memory on larger binaries, we used function-by-function analysis. BASICS enumerates functions with local stack frames, analyses each function as an entry point, and records whether the analysis is clean or produces a candidate report.

The evaluated projects cover 143,337 lines of C/C++ code, 55 binaries, and 15,713 analysed function entries. Table 7 summarises the analysis results, candidate filtering, and runtime measurements. The initial function-by-function sweep produced 7 candidate functions, all in SOEM. The isolated RBP-integrity filter removed one candidate (14.3%), leaving 6 functions in one binary for contextual review.

The isolated RBP-integrity filter removes candidate functions whose only evidence is a saved-frame-pointer integrity violation, without an accompanying stack-boundary, return-address, canary, underwrite, or unsafe-write violation. In function-by-function mode, such reports are frequently caused by missing caller context or conservative stack-state modelling and are therefore excluded from contextual review.

A focused re-analysis of the six final SOEM candidate functions did not confirm a real vulnerability. Two SOEM binaries and four libplctag binaries did not complete analysis. Five cases reached the 1,800 s analysis timeout, while one SOEM binary terminated because of an out-of-memory error during symbolic execution. The remaining binaries completed successfully with an aggregate analysis time of 13 h 43m and a wall-clock time of 2 h 24m under parallel execution.

Across 55 binaries and 15,713 analysed functions, the OSS sweep produced only a small number of candidate reports while remaining tractable in aggregate wall-clock time. These results support Q4. by showing that function-by-function analysis allows BASICS to scale to medium-to-large scale real-world binaries without requiring whole-program verification.

8.6. Function-by-function analysis on Linux package sweep

We also evaluated BASICS's function-by-function analysis ability on binaries extracted from Linux packages. This sweep was designed to test breadth: many small and medium binaries from common command-line packages, followed by an additional set of less prominent packages that process structured inputs such as archives, images, documents, and network data.

As shown in Table 8, the Linux sweep covered 58 packages, 237 binaries, and 5753 analysed functions across a diverse set of real-world Linux utilities. The initial function-by-function sweep produced 259 candidate functions. Applying the same isolated RBP-integrity filter removed 258 candidates (99.6%), leaving one function in one binary for contextual review.

The low candidate rate suggests that the filtering and caller-confirmation pipeline substantially reduces the review burden introduced by function-by-function analysis. Follow-up validation did not confirm the Linux candidate as a real vulnerability. These results support Q4. by showing that BASICS can be scheduled across hundreds of real Linux binaries while remaining tractable in terms of total execution time and the volume of candidate-review.

[1] https://github.com/DaveGamble/cJSON
[2] https://github.com/libplctag/libplctag
[3] https://github.com/OpenEtherCATsociety/SOEM

8.7. Candidate review

When applied to large real-world binaries, BASICS operates in two stages. First, it performs a scalable function-by-function sweep, analysing each recovered function independently and flagging functions that violate one or more security properties as vulnerability candidates. This initial stage prioritises coverage and scalability, allowing large binaries to be analysed without requiring whole-program exploration. Second, each candidate that remains after the isolated RBP-integrity filter undergoes contextual validation, where the analysis is repeated using additional execution context, typically from the program entry point or from a direct caller function. A candidate is considered confirmed only if the same violated property reappears and the counterexample trace still reaches the originally reported function.

The open-source and Linux sweeps produced six and one final candidate functions after filtering, respectively. However, follow-up contextual validation did not confirm any of these candidates as genuine stack-based buffer overflow vulnerabilities. Consequently, no previously unknown vulnerabilities were identified in the evaluated software. Candidate reports were re-analysed from the program entry point when the candidate function was itself the analysis entry, and from direct caller functions otherwise.

The SOEM `slaveinfo` candidates were produced by isolated function-by-function analysis. The flagged functions include formatting helpers such as `dtype2string`, `otype2string`, `access2string`, and `SDO2string`, plus mapping routines such as `si_PDOassign` and `si_map_sdo`. Manual inspection showed that these functions use bounded formatting operations, for example `snprintf` with `sizeof`-bounded static buffers, rather than unchecked writes to stack buffers. They were flagged because isolated analysis starts these functions without their real caller state and with highly unconstrained inputs. In addition, some EtherCAT helper calls and loops could not be fully emulated, which caused conservative stack-state approximations and erroneous underflow or frame-integrity violations.

The Linux sweep produced one candidate in `net-tools/netstat`, reported at `sub_40ee38`. This was not a real function entry but an internal basic block inside a larger function. BASICS flagged it because the function-recovery heuristic treated the block as an independent function and analysed it without the prologue and caller context that establish the stack frame. Re-running the analysis on the containing function removed the violation. Thus, this candidate was a control-flow recovery artefact rather than a memory-safety bug.

These results answer Q5. Function-by-function analysis enables BASICS to scale to large binaries by reducing the cost of whole-program exploration. As expected, analysing functions in isolation may introduce imprecision when caller context, function boundaries, external-call effects, or loop behaviour cannot be fully reconstructed. The contextual validation stage eliminated all final candidates in our evaluation, demonstrating its effectiveness as a filtering step. Nevertheless, the results indicate that improvements in caller-context reconstruction and function-boundary recovery could further reduce the number of candidates requiring follow-up analysis in large real-world binaries.

9. Discussion and limitations

Threat Model and Memory Scope. BASICS is designed to detect stack-based memory-safety violations whose effects can be represented in the current `MemStaCe`. The model focuses on stack objects, stack-frame metadata, and writes that may corrupt adjacent stack state. This is sufficient for detecting stack-based buffer overflows and corruption of control-relevant stack data, but it does not cover vulnerabilities whose effects are confined to other memory regions, such as heap overflows, use-after-free, or double-free bugs. In addition, the current memory model mainly tracks writes and does not explicitly model memory reads. As a result, BASICS does not claim to detect every memory-safety or control-flow attack, but rather stack-memory violations that manifest as violations of the modelled security properties.

Extending the `MemStaCe` with heap objects, read tracking, and richer control-flow metadata would allow BASICS to express more complex security properties, including properties for heap vulnerabilities and attacks that depend on read/write interactions or control-flow hijacking. These extensions are outside the scope of the current work.

Bounded Analysis. BASICS is a bounded analysis. It limits loop exploration, recursion depth, and symbolic execution in order to keep verification tractable. As a result, vulnerabilities that require deeper exploration may be missed. This behaviour is visible in the Juliet evaluation, where deeper call chains and loop-dependent writes contributed to false negatives.

The analysis also depends on approximating the behaviour of loops and external library functions. In some cases, these approximations are too conservative and report writes that are not actually possible, while in other cases they miss relevant memory updates. Since BASICS builds on angr, improvements in angr's binary analysis and symbolic execution directly improve the precision of the generated memory-state space.

Patch Validation. BASICS provides automated patch generation together with automated bounded validation, but it does not prove full behavioural equivalence between the original and patched binaries. The Validator checks whether malicious inputs no longer trigger the observed vulnerable behaviour and whether selected benign and boundary inputs preserve the original exit code and standard output.

This provides bounded evidence that a patch removes the observed vulnerability without breaking the tested executions, but it does not guarantee that all program behaviours are preserved. Stronger validation would require a larger input corpora or equivalence-checking techniques, which are outside the scope of this work.

Compatibility with Existing Defences. BASICS operates on recovered binary control flow and stack-memory state, and is therefore compatible with common mitigation mechanisms such as stack canaries, ASLR, and DEP/NX during vulnerability detection. In particular, BASICS explicitly models stack-canary integrity as one of its security properties. However, the current patching approach relies on trampoline-based binary rewriting through E9Patch. As a result, strict control-flow integrity (CFI) enforcement may restrict or invalidate some runtime patch redirections, depending on the target environment and runtime policy.

Scalability and Caller Context. The main scalability limitation of BASICS is full-binary analysis. For larger programs, analysis starting from the program entry point can exhaust memory or hit the analysis timeout. To address this, BASICS supports function-by-function analysis, where functions are analysed independently.

This mode improves scalability, but it also removes part of the caller context. As a result, function arguments, pointer relationships, and stack state may be modelled imprecisely, which can produce false positives. The real-binary sweeps showed that these issues can arise from incomplete caller information, incorrectly recovered function boundaries, or conservative stack modelling. Future work should improve caller-context reconstruction and function recovery for large binaries.

10. Extensibility of BASICS

BASICS was designed to be extensible by users, with support for user-provided security properties, patch templates, and CWE vulnerability maps. The `MemStaCe` is generic enough to support other security-property checks over stack memory.

To verify custom security properties, users can write LTL formulas using the standard LTL operators and the query functions from Fig. 6 to inspect the `MemStaCe` model. Users may also implement additional

query functions when the existing ones are insufficient. The tool then automatically compiles the LTL formula and uses it in subsequent verification runs.

By default, user-defined security properties are not associated with any known vulnerability class. To classify them, BASICS uses a CWE vulnerability map that associates security properties with CWE classes. When new properties are added, users can map them to existing CWEs or add new map entries. This allows BASICS to report property violations using the corresponding CWE classification.

If a newly reported vulnerability should also be repaired, the patching module requires a corresponding patch template for the identified sink. Users can extend the set of supported sinks by providing new patch templates. These templates are written as small C code snippets that include the custom `stdlib.c` file provided by E9Patch. Each template must define a `void` function named `apply_patch`, which receives as arguments the registers used to pass parameters according to the System V calling convention.

Finally, since BASICS is open source (Ferreirinha and Medeiros, 2026), users can modify the tool directly to add support for additional architectural or memory-model features. For example, one could model the x86-64 red zone by extending the stack-frame model with the 128-byte region below the stack pointer.

11. Related work

Prior work relevant to BASICS spans three largely separate areas: binary-level vulnerability discovery, formal verification of program safety properties, and automated repair. Existing binary-analysis systems are effective at discovering bugs or providing runtime bug oracles, while formal verification and automated repair systems have more commonly been applied at the source-code level. BASICS combines these threads in a narrower setting: detecting stack-based buffer overflows in x86-64 binaries through explicit stack-memory properties and using the resulting counterexamples to guide binary patching. This section positions BASICS against recent binary-analysis and sanitisation systems, source-level model checkers and repair techniques, and binary-rewriting systems.

11.1. Vulnerability discovery in source and binary code

Buffer-overflow detection has been extensively studied at the source-code level, where analyses can exploit variable declarations, type information, array bounds, and source-level control-flow structure. For example, CorCA (Inácio and Medeiros, 2023) combines static slicing with dynamic testing to detect BOs in C programs. Delta Pointers (Kroes et al., 2018) encode bounds information into pointer representations and raise a fault when an out-of-bounds pointer is dereferenced. (Lhee and Chapin, 2002) extend the C compiler with explicit type information about buffers, enabling runtime detection of invalid memory operations. These approaches demonstrate the value of combining program analysis with runtime evidence, but they assume access to source code or compiler instrumentation.

BASICS targets the harder binary-only setting, where compilation removes much of this structure. Binary vulnerability discovery must recover or approximate control flow, data flow, memory objects, and calling behaviour from machine code. Several systems address this problem for different vulnerability classes and analysis goals. Arbiter (Vadayath et al., 2022) combines static and dynamic analysis to detect multiple classes of binary vulnerabilities. Vyper (Boudjema et al., 2020) uses concolic execution and analysis of sensitive memory zones for multiclass vulnerability detection. IntScope (Wang et al., 2009) lifts x86 binaries to an intermediate representation and combines taint analysis with symbolic execution to detect integer overflows, while other work uses pattern matching and dynamic symbolic execution for the same class of bugs (Zhang et al., 2015). Machine learning has also been applied to binary vulnerability discovery: VulHawk (Luo et al., 2023) and related neural approaches (Schaad and Binder, 2022) embed disassembled code and train models to identify vulnerable binaries or functions.

Recent binary-analysis systems further emphasise scalability, execution speed, or runtime bug oracles. cwe_checker (Barabosch and Enkelmann, 2024) performs static checks for several CWE classes over lifted binary representations. ZAFL (Nagy et al., 2021) improves binary-only coverage-guided fuzzing through compiler-style binary instrumentation. SymQEMU (Poeplau and Francillon, 2021) and SymFit (Qi et al., 2024) improve the performance of binary symbolic and concolic execution. BoKASAN (Cho et al., 2023) and MTSan (Chen et al., 2023) provide binary-only runtime memory-safety oracles through sanitisation, object recovery, and memory-safety instrumentation. Operation Mango (Gibbs et al., 2024) scales static binary data-flow analysis to firmware taint-style vulnerabilities.

Table 9 summarises how these systems compare to BASICS. Existing binary tools are strong at triage, fuzzing, symbolic execution, runtime checking, taint analysis, or exploit discovery. BASICS targets a narrower problem: stack BOs in x86-64 binaries. Its main distinction is that it connects an explicit stack-safety violation to a binary patch and validation step.

11.2. Formal analysis of software security properties

Formal methods have been applied to software security through model checking, abstract interpretation, symbolic execution, and temporal-property checking. Source-level verifiers such as SLAM (Ball et al., 2004), BLAST (Beyer et al., 2007), CPAchecker (Beyer and Keremoglu, 2009), CBMC (Clarke et al., 2004), and SeaHorn (Gurfinkel et al., 2015) check reachability, assertion safety, pointer safety, and related safety properties over C programs or compiler-level representations. MOPS (Chen and Wagner, 2002) is closer to security-property checking: it models C programs and security rules as automata and was later used to find security flaws in large UNIX applications (Chen et al., 2004). These tools show the value of explicit specifications, but they rely on source-level structure that is unavailable in the binary-only setting targeted by BASICS.

At the binary and low-level memory-corruption level, fewer systems combine formal reasoning with vulnerability discovery. HeapHopper (Eckert et al., 2018) uses bounded model checking and symbolic execution to discover exploit primitives in heap allocator implementations. BASICS follows the same general idea of restricting formal reasoning to a modelled memory component, but targets stack-based BOs in x86-64 binaries. It models stack frames and byte-level stack states, then checks temporal properties over return addresses, saved frame pointers, canaries, adjacent stack objects, and unsafe writes. Unlike source-level verifiers, BASICS does not assume source code; unlike HeapHopper, it connects the detected violation to binary patching and validation.

11.3. Automated repair and binary rewriting

Automated vulnerability repair has mostly been studied at the source-code level. SenX (Huang et al., 2019) generates source-code patches from vulnerability-specific safety properties and a vulnerability-triggering input. CPR (Shariffdeen et al., 2021) uses concolic execution to co-explore the input space and patch space, filtering out overfitting patches. Program Vulnerability Repair via Inductive Inference (Zhang et al., 2022) infers likely patch invariants from program states and uses them to synthesise vulnerability patches. These systems are relevant because, like BASICS, they connect vulnerability evidence to repair. However, they assume source code and source-level patch locations, whereas BASICS repairs the deployed binary artefact directly.

For binary rewriting, BASICS builds on E9Patch (Duck et al., 2020), a static rewriting framework for x86-64 binaries. E9Patch provides the

Table 9
Comparison of BASICS with representative binary-level vulnerability-analysis systems.

System	Target	Method	Primary contribution	Scope relative to BASICS
BASICS	Stack BOs in x86-64 Linux binaries	Bounded model checking of stack-memory properties with LTL; bounded concolic execution for selected calls and loops; binary rewriting for supported sinks	Links stack-property violations to bounded validation and binary patching	Focused on stack BOs; patching limited to supported sinks and recovered stack objects
CWE_Checker (Barabosch and Enkelmann, 2024)	Several CWE classes in binaries	Static binary analysis over a lifted IR	Practical multi-CWE binary triage	Detection and warning generation; no LTL stack model or automated repair
ZAFL (Nagy et al., 2021)	Crash discovery in binary-only programs	Binary instrumentation for coverage-guided fuzzing	Improves fuzzing speed and coverage without source code	Optimises bug discovery through crashes; does not specify or repair stack-safety properties
Arbiter (Vadayath et al., 2022)	Multiple binary vulnerability classes	Hybrid static and dynamic analysis around vulnerable sinks	Detects several bug classes in binaries	Detection only; no stack-specific model checking or automated repair
BoKASAN/MTSan (Cho et al., 2023; Chen et al., 2023)	Runtime memory-safety errors in binaries	Binary-only sanitisation, object recovery, and memory-safety instrumentation	Provides strong runtime bug detection during testing and fuzzing	Coverage-dependent runtime checking; no formal stack-property verification or repair
Operation Mango (Gibbs et al., 2024)	Firmware taint-style vulnerabilities	Static binary data-flow analysis	Scales taint analysis to firmware binaries	Targets taint propagation rather than stack-state verification; no patching
HeapHopper (Eckert et al., 2018)	Exploit discovery in heap implementations	Bounded model checking and symbolic execution over heap-operation sequences	Automatically discovers heap exploitation primitives	Focused on heap allocators rather than stack BOs or binary repair

low-level mechanism for inserting trampoline code without requiring complete control-flow recovery. BASICS supplies the analysis layer: it uses model-checking counterexamples to identify vulnerable stack writes, selects a patch template, inserts the patch through binary rewriting, and validates the patched binary on concrete inputs. This makes BASICS less general than source-level repair systems, but more suitable for settings where source code is unavailable.

In summary, existing research has mainly focused on detecting vulnerabilities at the source code level or has applied formal methods mainly to verify functional correctness rather than explicitly to identify security vulnerabilities. Furthermore, vulnerability detection approaches have neglected the crucial aspect of automated vulnerability remediation. Our proposed approach addresses this research gap by employing formal verification to systematically discover stack BOs directly in binary code and combines this detection with an automated binary repair mechanism.

12. Conclusion

This paper presented BASICS, a binary-level framework for detecting and mitigating stack-based buffer overflows through bounded model checking and concolic execution. BASICS verifies LTL stack-security properties over a Memory State Space recovered from the binary and uses counterexample traces to identify and patch vulnerable sinks. The evaluation showed high-precision detection on controlled benchmarks, achieving 86.4% precision on SARD, and 86.1% precision on Juliet. All generated patches removed the observed vulnerable behaviour on the tested executions, while 41 out of the 46 patched binaries also preserved the original exit code and standard output on benign and boundary inputs. The open-source and Linux package sweeps further showed that function-by-function analysis allows BASICS to scale to larger binaries, while also highlighting the remaining limitations of bounded symbolic execution and caller-context reconstruction.

CRediT authorship contribution statement

Luís Ferreirinha: Writing – review & editing, Writing – original draft, Validation, Software, Methodology, Investigation, Formal analysis, Conceptualization. **Ibéria Medeiros:** Writing – review & editing, Writing – original draft, Validation, Supervision, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

This work was partially supported by P2030 through project I2DT, ref. COMPETE2030-FEDER-00389100, an ITEA4 European project (ref. 22025), and by FCT through the LASIGE Research Unit, ref. UID/00408/2025, DOI https://doi.org/10.54499/UID/00408/2025.

Data availability

BASICS is an open source tool, and it is available at Ferreirinha and Medeiros (2026).

References

Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J., 2005. Control-flow integrity. In: ACM CCS.

Aggarwal, A., Jalote, P., 2006. Integrating static and dynamic analysis for detecting vulnerabilities. In: COMPSAC.

Ball, T., 1999. The concept of dynamic analysis. ACM SIGSOFT SEN.

Ball, T., Cook, B., Levin, V., Rajamani, S.K., 2004. SLAM and static driver verifier: Technology transfer of formal methods inside microsoft. In: Integrated Formal Methods. Springer Berlin Heidelberg.

Barabosch, T., Enkelmann, N.E., 2024. CWE_Checker. https://github.com/fkie-cad/cwe_checker.

Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R., 2007. The software model checker blast: Applications to software engineering. STTT.

Beyer, D., Keremoglu, M.E., 2009. CPAchecker: A Tool for Configurable Software Verification. Technical Report, School of Computing Science (CMPT), Simon Fraser University (SFU).

Boudjema, E.H., Verlan, S., Mokdad, L., Faure, C., 2020. VYPER: Vulnerability detection in binary code. Secur. Priv..

Chen, H., Dean, D., Wagner, D.A., 2004. Model checking one million lines of C code. In: NDSS.

Chen, J., Han, W., Yin, M., Zeng, H., Song, C., Lee, B., Yin, H., Shin, I., 2022. SYMSAN: Time and space efficient concolic execution via dynamic data-flow analysis. In: USENIX Security.

Chen, X., Shi, Y., Jiang, Z., Li, Y., Wang, R., Duan, H., Wang, H., Zhang, C., 2023. MTSan: A feasible and practical memory sanitizer for fuzzing COTS binaries. In: USENIX Security.

Chen, H., Wagner, D.A., 2002. MOPS: An Infrastructure for Examining Security Properties of Software. Technical Report, University of California at Berkeley.

Cho, M., An, D., Jin, H., Kwon, T., 2023. BoKASAN: binary-only kernel address sanitizer for effective kernel fuzzing. In: USENIX Security.

Clarke, E.M., Henzinger, T.A., Veith, H., Bloem, R., 2018. Handbook of Model Checking. Springer Publishing Company, Inc..

Clarke, E., Kroening, D., Lerda, F., 2004. A tool for checking ANSI-C programs. In: TACAS.

Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., 1998. StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX Security.

CybersecurityDive, 2025. FBI, CISA warn hackers abusing buffer overflow CVEs to launch attacks. https://www.cybersecuritydive.com/news/fbi-cisa–hackers-buffer-overflow/740072/.

De Moura, L., Bjørner, N., 2008. Z3: An efficient SMT solver. In: TACAS.

Debian Project, 2026. Debian Policy Manual: Binary packages. Debian Project. URL https://www.debian.org/doc/debian-policy/ch-binary.html. (Accessed 23 May 2026).

Department, T.U., 2025. CWE - 2025 CWE top 25 most dangerous software weaknesses. https://cwe.mitre.org/top25/archive/2025/2025_cwe_top25.html.

Duan, R., Bijlani, A., Ji, Y., Alrawi, O., Xiong, Y., Ike, M., Saltaformaggio, B., Lee, W., 2019. Automating patching of vulnerable open-source software versions in application binaries.. In: NDSS.

Duck, G.J., Gao, X., Roychoudhury, A., 2020. Binary rewriting without control flow recovery. In: PLDI.

Duret-Lutz, A., Renault, E., Colange, M., Renkin, F., Aisse, A.G., Schlehuber-Caissier, P., Medioni, T., Martin, A., Dubois, J., Gillard, C., Lauko, H., 2022. From spot 2.0 to spot 2.10: What's new? In: CAV.

Eckert, M., Bianchi, A., Wang, R., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2018. HeapHopper: Bringing bounded model checking to heap implementation security. In: USENIX Security.

Ferreirinha, L., Medeiros, I., 2026. BASICS. https://github.com/Singularitty/BASICS.

Gerth, R., Dolech, D., Peled, D., Vardi, M., Wolper, P., 1995. Simple on-the-fly automatic verification of linear temporal logic. LICS.

Gibbs, W., Raj, A.S., Vadayath, J.M., Tay, H.J., Miller, J., Ajayan, A., Basque, Z.L., Dutcher, A., Dong, F., Maso, X., Vigna, G., Kruegel, C., Doupé, A., Shoshitaishvili, Y., Wang, R., 2024. Operation mango: scalable discovery of taint-style vulnerabilities in binary firmware services. In: USENIX Security.

GNU and Fee Software Foundation, 1986. GDB: The GNU project debugger. https://sourceware.org/gdb/.

Gurfinkel, A., Kahsai, T., Navas, J.A., 2015. SeaHorn: A framework for verifying C programs competition contribution. In: TACAS.

Holzmann, G., 1997. The model checker SPIN. IEEE TSE.

Hu, Y., Zhang, Y., Gu, D., 2019. Automatically patching vulnerabilities of binary programs via code transfer from correct versions. IEEE Access.

Huang, Z., Lie, D., Tan, G., Jaeger, T., 2019. Using safety properties to generate vulnerability patches.. In: IEEE S&P.

Inácio, J., Medeiros, I., 2023. CorCA: An automatic program repair tool for checking and removing effectively C flaws. ICST.

Keenlab, 2022. BinAbsInspector. https://github.com/KeenSecurityLab/BinAbsInspector.

Kim, S., Kim, R.Y.C., Park, Y.B., 2015. Software vulnerability detection methodology combined with static and dynamic analysis. Wirel. Pers. Commun..

Kroes, T., Koning, K., Van Der Kouwe, E., Bos, H., Giuffrida, C., 2018. Delta pointers: buffer overflow checks without the checks. In: EuroSys.

Lhee, K.s., Chapin, S.J., 2002. Type-assisted dynamic buffer overflow detection. In: USENIX Security.

Lin, Z., Jiang, X., Xu, D., Mao, B., Xie, L., 2007. AutoPaG: towards automated software patch generation with source code root cause identification and repair. In: ASIACCS.

Liu, H., Silvestro, S., Wang, X., Duan, L., Liu, T., 2019. CSOD: Context-Sensitive Overflow Detection. In: CGO.

Luo, Z., Wang, P., Wang, B., Tang, Y., Xie, W., Zhou, X., Liu, D., Lu, K., 2023. VulHawk: Cross-architecture vulnerability detection with entropy-based binary code search. In: NDSS.

Meng, X., Miller, B.P., 2016. Binary code is not easy. In: ISSTA.

MITRE Corporation - Common Weakness Enumeration, 2025a. CWE-121: Stack-based buffer overflow. https://cwe.mitre.org/data/definitions/121.html.

MITRE Corporation - Common Weakness Enumeration, 2025b. CWE-787: Out-of-bounds write. https://cwe.mitre.org/data/definitions/787.html.

Monperrus, M., 2018. Automatic software repair: A bibliography. ACM Comput. Surv..

Mossberg, M., Manzano, F., Hennenfent, E., Groce, A., Grieco, G., Feist, J., Brunson, T., Dinaburg, A., 2019. Manticore: A user-friendly symbolic execution framework for binaries and smart contracts.

Mouzarani, M., Sadeghiyan, B., Zolfaghari, M., 2017. Detecting injection vulnerabilities in executable codes with concolic execution. In: ICSESS.

Nagy, S., Nguyen-Tuong, A., Hiser, J.D., Davidson, J.W., Hicks, M., 2021. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In: USENIX Security.

NIST, 2017. Juliet C/C++ 1.3. https://samate.nist.gov/SARD/test-suites/112.

NIST, 2021. Software Assurance Reference Dataset (SARD). https://www.nist.gov/itl/ssd/software-quality-group/samate/software-assurance-reference-dataset-sard.

Open Source Security Foundation, 2026. Compiler Options Hardening Guide for C and C++. Open Source Security Foundation, URL https://best.openssf.org/Compiler-Hardening-Guides/Compiler-Options-Hardening-Guide-for-C-and-C%2B%2B.html.

Poeplau, S., Francillon, A., 2021. SymQEMU: Compilation-based symbolic execution for binaries. In: NDSS.

Qi, Z., Hu, J., Xiao, Z., Yin, H., 2024. SymFit: Making the common (concrete) case fast for Binary-Code concolic execution. In: USENIX Security.

Russell, R.L., Kim, L., Hamilton, L.H., Lazovich, T., Harer, J.A., Ozdemir, O., Ellingwood, P.M., McConley, M.W., 2018. Automated vulnerability detection in source code using deep representation learning. arXiv:1807.04320.

Schaad, A., Binder, D., 2022. Deep-learning-based vulnerability detection in binary executables. In: FPS.

Sen, K., 2007. Concolic testing. In: ASE.

Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N., Boneh, D., 2004. On the effectiveness of address-space randomization. In: ACM CCS.

Shariffdeen, R., Noller, Y., Grunske, L., Roychoudhury, A., 2021. Concolic program repair. In: PLDI.

Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A., Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G., 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In: IEEE S&P.

Szekeres, L., Payer, M., Wei, T., Song, D., 2013. Sok: Eternal war in memory. In: IEEE S&P.

The Linux Kernel Organization, 2025. The Linux Kernel User-Space API Guide. URL https://docs.kernel.org/userspace-api/index.html.

Vadayath, J., Eckert, M., Zeng, K., Weideman, N., Menon, G.P., Fratantonio, Y., Balzarotti, D., Doupé, A., Wang, R., Hauser, C., Shoshitaishvili, Y., 2022. Arbiter: Bridging the Static and Dynamic Divide in Vulnerability Discovery on Binary Programs. In: USENIX Security.

Wang, W., Fan, M., Yu, A., Meng, D., 2021. BOFSanitizer: Efficient locator and detector for buffer overflow vulnerability. In: HPCC/DSS/SmartCity/DependSys.

Wang, T., Wei, T., Lin, Z., Zou, W., 2009. IntScope: Automatically Detecting Integer Overflow Vulnerability in X86 Binary Using Symbolic Execution. In: NDSS.

Wheeler, D., 2025. Flawfinder. https://dwheeler.com/flawfinder/.

Xu, S., Wang, Y., 2022. BofAEG: Automated stack buffer overflow vulnerability detection and exploit generation based on symbolic execution and dynamic analysis. Secur. Commun. Netw..

Yun, I., Lee, S., Xu, M., Jang, Y., Kim, T., 2018. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In: USENIX Security.

Zhang, Y., Gao, X., Duck, G.J., Roychoudhury, A., 2022. Program vulnerability repair via inductive inference. In: ISSTA.

Zhang, Y., Sun, X., Deng, Y., Cheng, L., Zeng, S., Fu, Y., Feng, D., 2015. Improving accuracy of static integer overflow detection in binary. In: RAID.

Luís Ferreirinha is a Ph.D. Student at Vrije Universitiet Amsterdam, conducting research in programming languages and systems security. He holds an M.Sc. in Informatics from the Faculty of Sciences of the University of Lisbon (FCUL). He participated in the ADMORPH project in activities related to detecting vulnerabilities in binaries. His research interests include programming languages, type systems, formal verification, systems security, and cryptographic proofs. More information about him at https://www.lpferreirinha.com/.

Ibéria Medeiros is an Associate Professor at the Faculty of Sciences of the University of Lisbon (FCUL), and an integrated researcher of the LASIGE research unit. Her research interests are software security, vulnerability and attack detection, code privacy and correction, and artificial intelligence applied to cybersecurity. She is the author of tools for software security and cybersecurity, and has been involved in international and national projects. More information about her at https://www.di.fc.ul.pt/~imedeiros/.