BOFT: Exploitable Buffer Overflow Detection by Information Flow Tracking

Muhammad Monir Hossain, Farimah Farahmandi, Mark Tehranipoor, and Fahim Rahman
Electrical and Computer Engineering, University of Florida, Gainesville, Florida 32611, USA
hossainm@ufl.edu, {farimah, tehranipoor, fahimrahman}@ece.ufl.edu

***Abstract*—Buffer overflow is one of the most critical software vulnerabilities with numerous functional and security impacts on memory boundaries and program calls. An exploitable buffer overflow, which can be directly or indirectly triggered through external user domain inputs, is of a greater concern because it can be misused during run-time for adversarial intention. Although some existing tools offer buffer overflow detection to certain extents, there are major limitations, such as, poor detection coverage and ad-hoc/manual verification efforts due to inadequate predefined executions for static analysis and substantially large input subspace for dynamic verification. In this paper, to provide program verification in static time with high detection coverage, we propose an automated framework for Exploitable Buffer Overflow Detection by Information Flow Tracking (BOFT). We achieve this goal following three steps – first, BOFT analyzes the usage of arrays, pointers, and vulnerable application programming interface (APIs) in the program code and automatically inserts assertions required for buffer overflow detection. Second, BOFT instruments the program with taints for direct and indirect information flow tracking using an extensive set of formal expressions. Finally, it symbolically analyzes the instrumented code for maximum coverage and provides the list of exploitable buffer overflow vulnerabilities. BOFT is evaluated on standard benchmarks from SAMATE Juliet Test Suite (NIST) with a successful detection of ~94.87% (minimum) of exploitable buffer overflows with zero false positives.**

***Index Terms*—Buffer overflow, formal verification, information flow tracking, LLVM IR, symbolic execution.**

I. Introduction

Among numerous security concerns in the software domain, buffer overflow alone accounts for approximately half of prevailing vulnerabilities [1]. It can cause integrity and confidentiality violations for address spaces in the system leading to arbitrary code executions, denial-of-service (DoS), unauthorized memory accesses, and secret data leakage. Numerous reasons provoke buffer overflow in a program, notably written in languages that offer low-level and direct memory access (e.g., C/C++). Among them, incorrect assumptions about the buffer size, API misuses, and unsafe API handling are most notable. During the program development phase, it is practically impossible to identify all potential buffer overflows due to the program's substantial size and a variety of corner cases. Additionally, an exploitable buffer overflow, which can be directly or indirectly triggered through external user domain inputs, causes a greater security concern as it can be misused during run-time for adversarial intentions.

For potential buffer overflow detection, current research mainly focuses on two approaches – (1) dynamic analysis [2], [3] and (2) static program analysis with formal verification [4], [5]. The dynamic approach requires an extensive testbench or user inputs to cover triggering conditions; otherwise, the vulnerabilities remain undetected. Static analysis, on the other hand, suffers from low detection coverage and false positives due to limited formal definitions of potential overflow conditions from various sources. Integrating information flow tracking (IFT) capability with the static formal verification methods [6]–[8] can offer better coverage since it can identify the untrusted inputs being used as the index of an array for which boundary checking should be performed. However, existing IFT-based approaches fail to provide implicit information flow and over/under-tainting leading to false or missed detection.

In this paper, we introduce an automatic static verification framework – **Buffer Overflow Detection by Information Flow Tracking (BOFT)** – to detect exploitable buffer overflow vulnerabilities. BOFT integrates IFT into existing formal verification approaches to examine the influence of untrusted inputs (coming from user domain) on the arrays, pointers, and array-manipulating APIs in the program under verification (in C language). Compared to the existing approaches, BOFT framework provides advantages in terms of (1) both explicit and implicit tainting and semantic values, (2) comprehensive formal expression for assertions, and (3) automatic instrumentation for both tainting and assertion insertion. Using a symbolic execution engine, BOFT provides comprehensive code coverage and detects exploitable buffer overflow. Our major contributions are – (1) development of an automated formal verification framework for detecting exploitable buffer overflow; (2) development of an extensive library of formal expressions for accurate implicit and explicit IFT; (3) automated instrumentation of the program under verification for IFT and assertion insertion; and (4) exhaustive evaluation of BOFT

The rest of the paper is organized as follows. In Section II, we briefly describe related work. In Section III, we introduce BOFT framework. In Section IV, we describe the implementation detail. In Section V, we evaluate BOFT with experimental results. Finally, we conclude in Section VI.

II. Related Work

Existing solutions to detect buffer overflow suffers from several limitations. For example, dynamic analysis methods heavily rely on run-time execution and triggering [9]. But, it cannot identify all buffer overflows due to poor code coverage, inadequate test vectors, lack of scalability, and significant memory and run-time overhead. Traditional static analysis approaches (e.g., [10]) target to reduce verification overhead, but can have poor detection coverage as well. Serebryany et al. [11] analyzed LLVM Intermediate Representation (IR) of a program to detect explicitly visible buffer overflow. Unfortunately, exploitable buffer overflow vulnerabilities, where external inputs trigger the overflow, can also be implicit in nature. IFT can identify the taint propagation of external triggers by instrumenting the code [12]. [6], [7] used dynamic IFT for specific inputs to detect buffer overflow. Manzano et al. [8] analyzed semantics usage for taint propagation for LLVM IR instructions. However, these proposed techniques overlook implicit flows (relating to control flow dependencies) which can cause major security

The authors would like to acknowledge funding support through Semiconductor Research Corporation (Grant/Award No. 2018-TS-2860 / AWD05064).

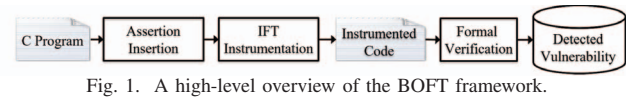

Fig. 1. A high-level overview of the BOFT framework.

breach [12]. Additionally, semantic values in the propagation must be considered for tainting (or untainting) any implicit flow for efficient detection.

III. BOFT Framework

A. Overview

In this work, we propose BOFT, an automated framework to detect exploitable buffer overflow using information flow tracking. In our framework, IFT confirms the misuse of arrays, pointers, and vulnerable APIs controlled by the untrusted input domain. On the other hand, formal verification provides the formal proof of the presence of buffer overflow.

A high-level overview of the BOFT framework is shown in Figure 1. At first, the program under verification (C code) is analyzed and necessary assertions are generated to detect exploitable buffer overflow. The program, now with required assertions, is instrumented for IFT to analyze exploitable variables (triggers) using explicit and implicit tainting. Finally, we perform formal verification on the instrumented code through a symbolic executor (KLEE) using a formal Satisfiability Modulo Theories (SMT) solver in the background. BOFT performs the assertion insertion, IFT instrumentation, and taint propagation analysis automatically (see Section IV). Below we discuss assertion formulation and IFT instrumentation strategies used in BOFT implementation.

B. Assertion Formulation for Buffer Overflow Detection

To detect the buffer overflow vulnerabilities, we identify the locations where buffers are being used, modified, or accessed. BOFT explores all possible sources, such as, (1) code constructs of an array, (2) code constructs of a pointer, and (3) unsafe APIs in C-language. An exploitable buffer overflow occurs when it reads/writes out of boundary through an exploitable index or makes an out of bound array access through a pointer linking to an untrusted array. This type of exploitable buffer overflow can be identified using the following formal expression of taint assertions.

$$\forall M[x] \notin [R_{min}, R_{max}] \wedge Taint(x) \rightarrow BOVF(M[x]) \quad (1)$$

where $M[x]$ is memory access at x index in buffer, R_{min} and R_{max} are lower and upper bound of buffer, respectively, $Taint(x)$ returns whether index x is tainted, and $BOVF(M[x])$ is detected buffer overflow for memory access.

Additionally, string manipulating APIs [13] from C-language usually operates on the source and destination buffers. The following expression represents such buffer overflow cases.

$$Taint(Src) \wedge (Size(Src) > Size(Dest)) \rightarrow BOVF \quad (2)$$

where $Dest$ and Src represent the destination and source buffers, respectively.

In all cases, buffer overflow takes place due to the lack of a boundary checking in the program under verification. When a user input can control the buffer size and perform data manipulation, the attacker (from user domain) can exploit such inputs to overflow the buffer. Therefore, generally, two types of assertions are sufficient to cover most of the exploitable buffer overflows. The first one is to check the taint status of the array index, pointer, and arguments (which is used as a reference to another array and pointer in a subroutine) in the vulnerable API, as discussed previously. It allows us to extensively identify exploitable buffer overflow cases and significantly reduce false detection. The other type of assertion is for boundary checking, i.e., identifying boundary cases for the usage of tainted variables. This provides the formal proof of potential illegitimate usage of arrays and pointers.

C. Integrating Information Flow Tracking

In a program, an input that can import data from the user domain is exploitable, hence, is untrusted. To identify exploitable buffer overflow, we must identify and track these inputs (explicit flow tracking) or their logical/arithmetic implications on program control flow (implicit flow tracking). Using IFT, we can taint the variables and analyze the propagation of the taint. This allows us to precisely set bound checking assertions for buffer overflow detection. However, it requires an instrumentation of the code with associated IFT logic (or assertion). For an efficient IFT integration, we take account of three crucial factors – (1) identification of all types of taint sources in the program; (2) analysis of taint propagation logic, i.e., instructions to validate explicit or implicit information flow from its source to destination operand; and (3) scopes of exploitation due to untrusted inputs are also known as *sinks*.

1) Taint Sources: We consider various external input sources as tainted variables such as command arguments, standard input APIs, and File I/O stream in our BOFT framework as the attackers can provide malicious data through these interfaces. Again, tainted data may traverse from one function to another in a program either as an argument or return variable, which are also considered as taint sources in BOFT.

2) Taint Propagation Logic: Once untrusted variables are tainted, BOFT must analyze each program's instructions on how these tainted variables interact with other variables, explicitly or implicitly, to cause a buffer overflow. We consider the following four general categories of instructions to perform this analysis.

Data Transfers: The destination data would be tainted if any byte of the source data is tainted. If the operands are literals, it does not taint the result. Because the source of literals is either from hard-coded data in the program or assigned while compiling the program.

Arithmetic and Logic Operations: If any byte of the source operand is tainted, the result will only be tainted when it is affected by the operand. When tainted variables are overridden by a dominant variable (e.g., trusted variables, constant, etc.), tainting should be omitted to avoid unnecessary over-tainting. For example, the result should not be tainted when a tainted variable with the value of zero adds with a trusted variable.

Control Flow: Control instructions, such as conditional branches and jumps, are generally affected by implicit tainting. For example, when any operand related to the conditional branch instruction is tainted, BOFT taints the resultant variable. The implicitly tainted variable here does not have any direct connection to the untrusted input.

Floating Point: Since floating-point instructions basically perform arithmetic operations, we consider the tainting strategy similar to arithmetic and logical instructions for double and single precision formats.

3) Sinks: Identifying sinks helps to determine how the tainted data may be used maliciously in the code, for example, illegitimate overwriting of stack contents resulting into security vulnerabilities. BOFT identifies and analyzes such usages in the program for precise detection of the vulnerability location.

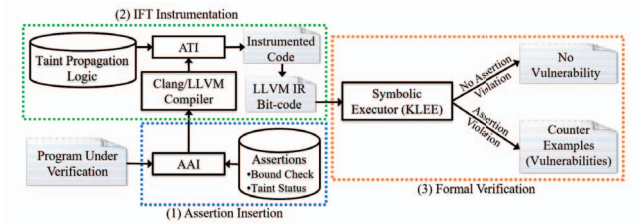

Fig. 2. Implementation workflow of BOFT.

IV. BOFT Workflow

In this section, we discuss the workflow of BOFT framework. As shown in Figure 2, it consists of three major operational stages – (1) assertion insertion, (2) IFT instrumentation, and (3) formal verification, as described below.

A. Assertion Insertion in Automatic Fashion

In this step, we develop the *automatic assertion insertion (AAI)* tool that takes the program under verification and inserts the boundary and taint checking assertions. AAI is developed in C++, utilizing the `Clang-AST Libtooling` library. AAI analyzes the program to find all usages of arrays, pointers, and unsafe APIs following the pseudo-algorithm presented in Algorithm 1. First, it takes the program as an input and generates the abstract syntax tree (AST). The procedure `Matcher` accepts the defining code structures and invokes a callback function when it finds an AST match. As mentioned in section III-B, we require boundary and taint checking assertions to detect exploitable buffer overflow. Hence, from the returned AST node, AAI extracts all declared variables, sizes, etc., relevant to the array, pointer, and API to provide the boundary assertion and the identifiers for taint checking assertion. It also defines the untrusted sources as symbolic for formal verification. The outcome of AAI is the instrumented program with assertions.

B. IFT Instrumentation

In this step, the assertion inserted code is translated into LLVM IR for IFT instrumentation. The advantages of this transformation is twofold – (a) reduced instrumentation complexity due to its limited syntax compared to the C language, and (b) flexibility to transform LLVM IR into any desired platform for final deployment after verification. To do so, first, we develop a comprehensive set of expressions for accurate and efficient taint propagation for both explicit and implicit flow.

Algorithm 1: Automatic Assertion Insertion (AAI).

Input: User C Coding File, $FILE$
$AST \leftarrow Generate\_AST(Compiled(FILE))$
$Matcher(AST, Array, @CallBack(Arr))$ /* Array Usage */
$Matcher(AST, API, @CallBack(API))$ /* Vulnerable API Usage */
$Matcher(AST, Pointer, @CallBack(Ptr))$ /* Pointer Usage */
$CallBack(Arr) : Arr[i++] \leftarrow (ArrId, IndexId, LineNum)$
$CallBack(API) : API[j++] \leftarrow (ArgId, ArgSize, LineNum)$
$CallBack(Ptr) : Ptr[k++] \leftarrow (PtrId, IndexCount, LineNum)$
while $EOF(FILE) = FALSE$ **do**
  $Line\_Num \leftarrow ReadLine(FILE)$
  if $Line\_Num = Array[i].LineNum$ **then**
    insert $Declare\_Symbolic\_Var(Array[i])$
    insert $Boundary\_And\_Taint\_Assertion(Array[i++])$
  end
  if $Line\_Num = API[j].LineNum$ **then**
    insert $Declare\_Symbolic\_Var(API[j])$
    insert $Boundary\_And\_Taint\_Assertion(API[j++])$
  end
  if $Line\_Num = Pointer[k].LineNum$ **then**
    insert $Declare\_Symbolic\_Var(Pointer[k])$
    insert $Boundary\_And\_Taint\_Assertion(Pointer[k++])$
  end
end
$out\ FILE_{Instrumented}$

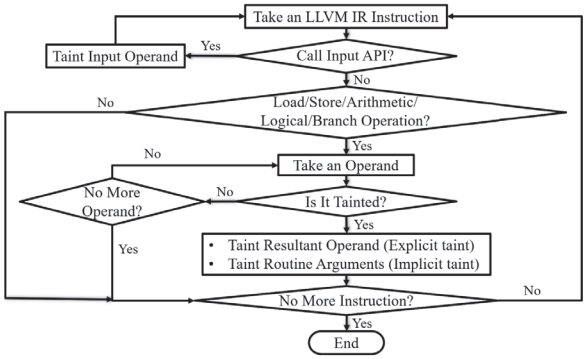

Fig. 3. Flowchart of explicit and implicit taint propagation.

Then, we develop *automatic taint instrumentation (ATI)* tool to automatically instrument the code with these expressions.

1) Formal Expressions for Taint Propagation: In Table I, we list some example expressions for taint propagation. For instance, in the case of `add` instruction, taints of operands are defined as t_{op1} and t_{op2}. The taint for the destination operand depends not only on the source operands but also on the values. The destination operand is only tainted when either of the source operands is nonzero and tainted. This library is a critical requirement for identifying all exploitable buffer overflow vulnerabilities in a program.

2) Automatic Taint Instrumentation (ATI): We develop ATI in C# for both implicit and explicit IFT. Figure 3 provides the high-level flow of the approach. It takes all four types of instructions into account to identify if there is any untrusted operand that should be tainted or is already tainted by a previous operand. It taints the resultant operands following Table I. For explicit taint propagation, the taint propagates from the source operands to the destination operand. However, for the cases of control instructions, such as `branch`, ATI needs to execute implicit tainting. For this, when a `branch` instruction is found tainted, all corresponding assignment statements of the branch are tainted. When some nested branches cause implicit tainting, we insert the taint logic that if one of the branches allows implicit taint propagation, the corresponding assignment statements are also tainted without executing the instructions for the taint logic for remaining branch instructions. As a result, the number of executed instructions is reduced significantly.

C. Formal Verification

The final stage of BOFT framework provides the formal proof of vulnerabilities in the program. It utilizes an existing symbolic execution engine, KLEE to run the compiled binary of the instrumented LLVM IR code. It traverses all paths symbolically and propagates the taints from the sources to the resultant operand. Using the taint and boundary checking assertions instrumented in the previous stages, KLEE generates constraints based on those assertions and utilizes a SMT solver (Z3) to validate the assertions. Once an assertion violation with a counter-example is generated, BOFT can detect corresponding exploitable buffer overflow.

V. Experimental Results and Evaluation

In this section, we first present an example case study, followed by the experimental results on several benchmarks to evaluate our BOFT framework. All experiments are performed on a VMWare Linux machine configured with one core of Intel i7-9700K 3.6GHz CPU and 12 GB RAM. We use KLEE v2.0, SMT solver Z3 v4.8.8, and compiler Clang LLVM v6.0.

TABLE I
EXAMPLE TAINT PROPAGATION EXPRESSIONS FOR LLVM IR INSTRUCTION SET

Ins.	Syntax	Expression of Taint Propagation	Taint Type
add	$< result >= add < ty >< op1 >, < op2 >$	$((op1 \neq 0) \wedge t_{op1}) \vee ((op2 \neq 0) \wedge t_{op2})$	Explicit
mul	$< result >= mul < ty >< op1 >, < op2 >$	$((op1 \neq 1) \wedge t_{op1}) \vee ((op2 \neq 1) \wedge t_{op2})$	Explicit
load	$< result >= load[volatile] < ty >, < ty > * < pointer >$	$t_{pointer}$	Explicit
store	$store[volatile] < ty >< value >, < ty > * < pointer >$	t_{value}	Explicit
br	$bri1 < cond >, label < iftrue >, label < iffalse >$	t_{cond}; *Taint all assignments for label $< iftrue >$	Implicit
icmp	$< result >= icmp < cond >< ty >< op1 >, < op2 >$	$t_{op1} \vee t_{op2}$	Implicit

TABLE II
BENCHMARK RESULTS AND COMPARISON

Vulnerability ID	Vulnerability Definition	#Test Cases	Frama-C				KLEE (without instrumentation)				BOFT			
			Successful Detection	False Positive	Avg. run-time	Avg. Code Size (bin)	Successful Detection	False Positive	Avg. run-time	Avg. Code Size (bin)	Successful Detection	False Positive	Avg. run-time (Assertion Violation)	Avg. Instrumented Code Size (bin)
CWE121	Stack Overflow	10	3	3	0.30 s	1x	2	0	0.32 s	1x	9	0	0.72 s	1.62x
CWE122	Heap Overflow	10	3	1	0.27 s	1x	1	0	0.26 s	1x	10	0	0.67 s	1.79x
CWE124	Buffer Underwrite	7	1	0	0.32 s	1x	1	0	0.30s	1x	7	0	0.71 s	1.68x
CWE126	Buffer Over-read	7	2	2	0.26 s	1x	1	0	0.28s	1x	6	0	0.65 s	1.72x
CWE127	Buffer Under-read	5	1	0	0.29 s	1x	1	0	0.27 s	1x	5	0	0.64 s	1.65x

A. Example Case Study

A simplified code based on the vulnerability ID CWE-121 [14] is shown in Listing 1.

Listing 1. Case Study: Detection of Exploitable Buffer Overflow.

```
1 int main(void){
2         int var_a, var_b, int buffer[10];
3         scanf("%d", &var_b);
4         var_a = var_b - 1;
5         for (int i = 0; i <= var_a; i++)
6             buffer[i] = i;
7         return 0;
8 }
```

Information Flow: The user provides input via `scanf`, which taints directly `var_b`. At line 4, an explicit taint propagates from `var_b` to `var_a` through the arithmetic operation. Another explicit tainting occurs at line 5, (`i++`). Tainted variable, `var_a`, taints buffer index, `i`, implicitly in the loop condition. As the adversary can control the loop iterations and write data into the buffer, this shows an exploitable buffer overflow.

Vulnerability Detection: When we verify this piece of code using BOFT, it automatically instruments the code and generates the instrumented LLVM IR with two assertions ($taint(i) = 0$ and $i < SIZE$) and IFT logic. During verification, each assertion is *violated*. Thus, BOFT has identified the exploitable buffer overflow vulnerability for the given example.

B. Benchmark Results

To evaluate BOFT, we have used 39 test cases from SAMATE Juliet benchmark suite from NIST [15] based on 5 CWE IDs. We have instrumented the test cases for intaking external (untrusted) user inputs so that exploitable buffer overflow can be analyzed. The evaluation results focus on the number of successful detection, false alarm, and run-time and memory overhead, as summarized in Table II.

Detection Coverage: As we see from Table II, BOFT can successfully detect 37 out of 39 cases, with an average of ~94.87%, while showing no false positive. This is because BOFT incorporates both bound checking and taint status checking assertions, accurately and efficiently instrumented with taint propagation logic, and finally, leverages symbolic execution to cover all exploitable buffer overflow in the program.

Comparison with Frama-C and KLEE: For the same test cases, Frama-C [16] and KLEE provide ~25.64% and ~15.38% detection capability with ~15.38% and %0 false positives, respectively. Frama-C and KLEE were able to identify some explicit overflow cases only. Since, for an exploitable buffer overflow, the inputs are dynamic and IFT analysis is needed to detect variables causing out of bound accesses, static analysis tools like Frama-C cannot detect them.

Overhead Analysis: On average, BOFT takes ~$2.3x$ verification run-time compared to Frama-C and KLEE. BOFT leverages KLEE Symbolic Engine to verify all candidate assertions to achieve formal proof of vulnerabilities. Hence, it traverses all potential control paths resulting in a higher verification run-time. Additionally, the instrumented binary size is ~1.69 times of the uninstrumented program under verification. It is understandable since BOFT relies on extensive instrumentation of the code for taint propagation and assertion insertion. However, the increased code is only for verification phase, finally the deployed program is not affected due to this instrumentation.

VI. CONCLUSION AND FUTURE WORK

We have presented BOFT, a framework for automatic detection of exploitable buffer overflow vulnerabilities. BOFT integrates formal verification with IFT and leverages symbolic execution. Using this combined approach, over ~94% of vulnerabilities can be detected with zero false positives. This work's future direction includes an extensive library of taint logic preparation, comprehensive evaluation on industry-scale benchmarks, and platform integration.

REFERENCES

[1] https://www.synopsys.com/blogs/software-security/top-10-software-vulnerability-list-2019/.
[2] H. Liu *et al.*, "Csod: context-sensitive overflow detection," in *Int. Symp. on Code Generation and Optimization (CGO)*, 2019, pp. 50–60.
[3] A. Slowinski *et al.*, "Body armor for binaries: preventing buffer overflows without recompilation," in *USENIX ATC* 12, 2012, pp. 125–137.
[4] G. Brat *et al.*, "Ikos: A framework for static analysis based on abstract interpretation," in *Int. Conf. on SW Eng. and Formal Methods*, 2014.
[5] A. Gurfinkel *et al.*, "The seahorn verification framework," in *Int. Conf. on Computer Aided Verification*, 2015, pp. 343–361.
[6] J. Clause *et al.*, "Dytan: a generic dynamic taint analysis framework," in *Int. Symp. on SW Testing and Analysis*, 2007, pp. 196–206.
[7] M. G. Kang *et al.*, "Dta++: dynamic taint analysis with targeted control-flow propagation." in *NDSS*, 2011.
[8] R. Corin and F. A. Manzano, "Taint analysis of security code in the klee symbolic execution engine," in *Int. Conf. on Information and Communications Security*, 2012, pp. 264–275.
[9] E. Haugh and M. Bishop, "Testing c programs for buffer overflow vulnerabilities." in *NDSS*, 2003.
[10] N. Hasabnis *et al.*, "Light-weight bounds checking," in *Int. Symp. on Code Generation and Optimization*, 2012, pp. 135–144.
[11] K. Serebryany *et al.*, "Addresssanitizer: A fast address sanity checker," in *USENIX Annual Technical Conference*, 2012.
[12] R. Zhang *et al.*, "Combining static and dynamic analysis to discover software vulnerabilities," in *Int. Conf. on Innovative Mobile and Internet Services in Ubiquitous Computing*, 2011.
[13] R. C. Seacord, *The CERT C secure coding standard*, 2008.
[14] https://cwe.mitre.org/.
[15] https://samate.nist.gov/SARD/testsuite.php.
[16] P. Cuoq *et al.*, "Frama-c," in *Software Engineering and Formal Methods*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 233–247.