Computer Security Symposium 2020
26 - 29 October 2020

PRSafe: Primitive Recursive Function based Domain Specific Programming Language using LLVM

Sai Veerya Mahadevan[1,a)] Yuuki Takano[1] Atsuko Miyaji[1,2]

Abstract: General Purpose programming languages such as C++, Python suffer from resource management and input errors because they are Turing-complete. These languages assume the properties of a Turing machine, with infinite memory and computational power. For most real world scenarios however, Turing-complete programs are unnecessary. The goal of this paper is to introduce a prototype functional Domain Specific Language (DSL) called PRSafe. The design of PRSafe is based on the memory and input properties of Primitive Recursive Functions (PRFs). Hence, all computations must terminate. For the language implementation, we use the Lex and Bison softwares for lexing and parsing respectively and LLVM (Low-Level Virtual Machine) compiler infrastructure for code generation and optimization. PRSafe is designed such that syntactically and semantically, it prohibits the creation of Turing-complete programs. It is eventually intended to be a mathematically verifiable, friendly compiler with better error-diagnostics for eBPF programs than the eBPF verifier.

Keywords: programming language, primitive recursive function, LLVM, eBPF

1. Introduction

A language compiler was originally looked at as a black-box, which accepted some High level language program as input, performed some invisible processing on this input and generated some machine code which is platform specific, as output. However, over the last 20 years, with the introduction of the LLVM Compiler Infrastructure project [1], the compiler has become more analyzable. Using LLVM, almost any High level language program can be processed to obtain a readable, descriptive Intermediate Representation (IR) known as LLVM IR. This LLVM IR is platform agnostic. Also, any compiler level optimizations can be performed on this IR, such as constant folding [2] or 'Dead-code elimination'. This benefits the user program, independent of the platform it is run on. Due to these features of LLVM, it is used extensively to create Domain Specific Languages (DSLs).

The domain of this project is the eBPF tool. The extended Berkeley Packet Filter (eBPF), an extension of the Berkeley Packet Filter (BPF)[3], is an in-kernel virtual machine that allows user-space programs to attach to specific hooks in the kernel and run safely. In eBPF, programs are passed to LLVM to generate eBPF bytecode. This bytecode is passed to the eBPF verifier, a module that only allows programs with a specific feature-subset of C, to run inside the BPF VM. This ensures the safety property. Some of the checks that the eBPF verifier performs include verifying if the program forms a Directed Acylic Graph (DAG), ensuring that no loops are allowed. The strictness of the eBPF verifier albeit essential for safety, puts the onus on the average network programmer to learn to write only eBPF verifier approved programs, even before generating bytecode.

To simplify the burden on the user, this project aims to provide a user-friendly pre-verifier to the eBPF verifier. PRSafe is intended to accept user written programs that semantically adhere to the requirements of the eBPF verifier, generate LLVM IR, to allow the user to leverage the optimization methods provided by LLVM, and finally generate bytecode that is acceptable by the eBPF verifier. Thus, it intends to provide a time-saving, syntactic and semantic check to verify that programs are acceptable by the eBPF verifier, even before bytecode generation is done and passed to it.

PRSafe, is based on the memory and input safety properties of Primitive Recursive Functions (PRFs) and hence, adhere to the safety properties assured by PRFs.

2. Background and Related Work

In this section we discuss the tools behind language design, the drawbacks of a General-purpose programming language, the goals to be met while designing a new Domain Specific Language and the properties of Primitive Recursive functions.

[1] Osaka University
[2] Japan Advanced Institute of Science and Technology
[a)] saiveerya@cy2sec.comm.eng.osaka-u.ac.jp

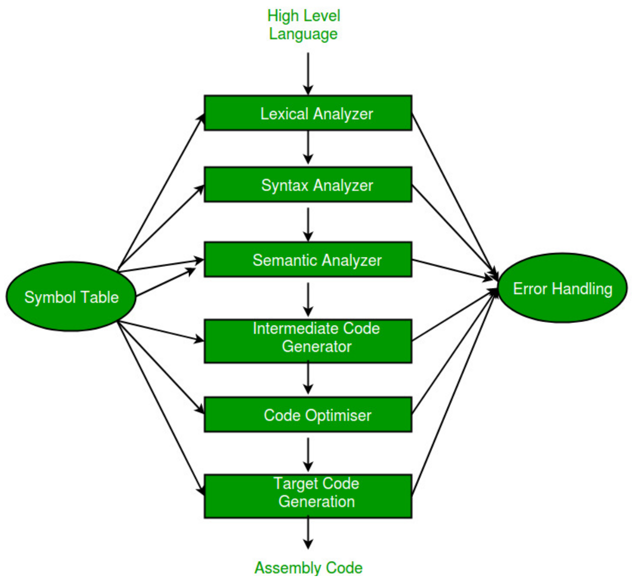

Fig. 1 Phases of a standard compiler

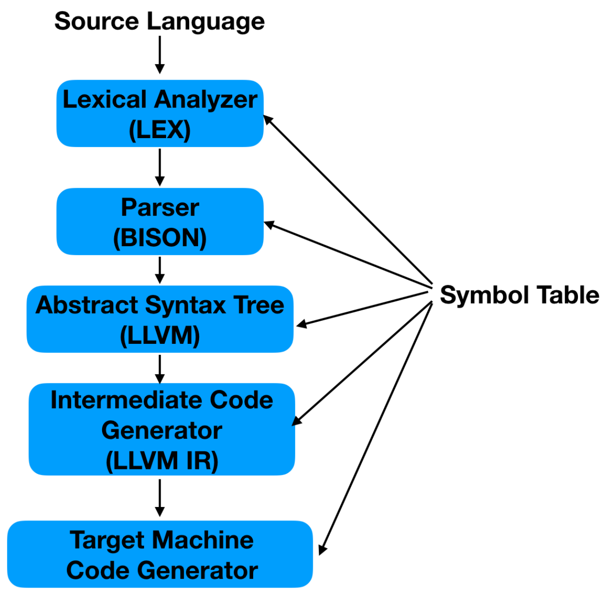
Source Language
Lexical Analyzer (LEX)
Parser (BISON)
Abstract Syntax Tree (LLVM)
Intermediate Code Generator (LLVM IR)
Target Machine Code Generator
Symbol Table

Fig. 2 Phases of PRSafe compiler

2.1 Tools used to design PRSafe

The standard phases of a compiler are split into roughly 4 modules that include a Lexer, Parser, AST, Code Generator for a target machine architecture. Phases of a compiler is clear in Figure 1.

[4] For PRSafe, the structure of the compiler is in Figure 2.

The first module is the lexical analyzer or lexer. The PRSafe lexer accepts the source language input only if contains a permitted set of tokens, including user-defined terms such as identifiers, basic data types, operators. The lexer is written in the Lex software. The tokens recognized by the lexer module are then passed as input to the second module, the parser. The parser implements the Extended Backus–Naur form grammar (EBNF) specified for PRSafe using Bison software. The grammar is described in detail in Section 5. Next, the statements in the source language are organised into an Abstract Syntax Tree (AST) written in LLVM using an AST module. Finally, code-generation is performed by the LLVM based code-generator module using the AST generated as a program structure template.

2.1.1 Introduction and Advantages of LLVM : The Compiler Infrastructure

Any Programming language falls under two broad categories by method of compilation - compiled language (C++) vs interpreted language (Python). The major difference is that a compiled language compiles its source code to machine code directly translated on the target machine, whereas an interpreted language does a line-by-line code translation. This leads to some obvious differences between the two methods of compilation. For example, a compiled language offers better performance but cannot accept run-time changes, whereas an interpreted language addresses its debugging issues only at run time.
This is where LLVM, is beneficial. Among other features of LLVM, it provides some features that combine those of compiled and interpreted languages.
Some features of LLVM used in PRSafe:

- PRSafe accepts a C program input and performs code-generation by generating LLVM IR. This IR is used for code analysis, optimization and new feature inclusion.
- PRSafe uses the LLVM Just-in-Time (JIT) compiler. This implies that LLVM converts source language input to an LLVM bytecode, rather than a regular compiled or interpreted language.
- In PRSafe, LLVM uses a symbol table to ensure a static, semantic restrictions, i.e. keeping track of identifiers and variables scope, type.

2.2 Turing Complete vs. Non-Turing complete Languages

In this section, we describe the drawbacks of General-purpose programming languages.

2.2.1 Turing Complete Language

Structured programming languages such as C, C++, Python etc. are Turing complete in nature. i.e. these languages use the model of a Turing machine, a machine which has an infinite memory that can run any computable functions of a program of a finite length; based on program inputs and constructs, it is determined what the program will do next. Although the general notion is that Turing complete languages are the basic requirement for programming language design, this is not necessarily true. Owing to the safety-threatening properties of Turing complete language, a number of Domain Specific Languages (DSL) have been designed.

Fundamental safety-threatening features of a Turing Complete Language:

- Unbounded memory ; This means an infinite memory can be used
- Infinite loops

An example of an unsafe function that can be written in a Turing language, with a logical error.

```
int a=0;
while(1):
{
        a++;
       if(a > 0)
        printf(a);
}
```

This leads to a non-terminating loop. If this function had authorized access to the kernel space in a machine, an unauthorised user could not only perform an unsafe, infinite computation but also deplete a huge amount of computational resources in the process.

2.2.2 Non-Turing complete Languages

In contrast, we have Non-Turing complete Languages. A non-Turing complete language essentially ensures the following properties:

- Does not support infinite loops
- Random accesses to memory
- Types of Recursion such as Mutual recursion, Multiple Recursion
- Jump statement and variants.

2.3 Primitive Recursive Functions

2.3.1 What is Primitive recursion?

Primitive recursion is a scheme that permits to define a function by descending recursion in one variable, or to phrase it in the imperative paradigm: it has for-loops as the only iterative control structure. Therefore primitive recursive programs always terminate [5]. To eliminate the unsafe properties associated with a Turing complete language, while ensuring computational power, we rely on a set of functions called Primitive Recursive Functions (PRF). Primitive Recursive Functions can be used to express almost all real-life application mathematical functions.

2.3.2 Expressing a PRF

A Function is considered primitive recursive if it can be obtained from some specific initial functions (Zero Function, Successor Function and Projection Rule) and through finite number of composition and primitive recursion steps.

The two powerful functions to derive PRF are

- Composition Rule
- Recursion rule

2.3.3 Composition Rule

If a function f is defined as the composition of (previously defined) primitive recursive functions i.e.

$$if\, g_1(x_1,\ldots,x_n), g_2(x_1,\ldots,x_n),\ldots,g_k(x_1,\ldots,x_n) \tag{1}$$

are primitive recursive and

$$h(x_1,\ldots,x_k) \tag{2}$$

is primitive recursive, then

$$f(x_1,\ldots,x_n) = h(g_1(x_1,\ldots,x_n),\ldots,g_k(x_1,\ldots,x_n)) \tag{3}$$

2.3.4 Recursion rule

If a function f is defined by recursion of two primitive recursive functions, i.e.

$$if\, g(x_1,\ldots,x_{n-1})h(x_1,\ldots,x_{n+1})$$

are primitive recursive , then the following function is also primitive recursive

$$f(x_1,\ldots,x_{n-1},0) = g(x_1,\ldots,x_{n-1}) \tag{4}$$

$$\begin{gathered} f(x_1,\ldots,x_{n-1},m+1) = \\ h(x_1,\ldots,x_{n-1},m,f(x_1,\ldots,x_{n-1},m)). \end{gathered} \tag{5}$$

2.3.5 Example of a PRF:

The exponentiation function exp(x, y) = x y is primitive recursive. Explanation: We can define exp() primitive recursively as

$$exp(x,0) = 1 \tag{6}$$

$$exp(x,y+1) = mult(x,exp(x,y)). \tag{7}$$

where

$$\begin{aligned} f(x) = succ(zero(x)) = 1, \\ g(x,y,z) = x \cdot z \end{aligned} \tag{8}$$

2.4 Related Work

By virtue of the restricted functionality in a Primitive Recursive Function based Language, many languages have been designed for purely theoretical discussion or academic exploration. Three such languages used in Academia are (i) BlooP [6], (ii) Exanoke - a pure functional language only to express PRFs [7] (iii) Agda- a dependently typed functional programming language [8]

The grammar of these languages ensure bounded loops, lack forms of recursion and focus on type safety. The basic design rules of construction of simple languages is very similar to PRSafe, however it differs in the intent of application and hence infrastructure tools used for implementation of these languages.

2.5 Overview

The combined power of the safety provided by Primitive Recursive Functions (PRFs) and the re-usability offered by LLVM, allow us to ensure the memory-bound and time-bound consumption of resources while also offering the flexibility to increase or restrict the type of input accepted and thus inherently implement properties like type-safety.
Primitive Recursive Functions can be used to express almost all mathematical functions, however we can use the recursion properties of PRFs to restrict the scope of application to functions that definitely terminate.
When it comes to LLVM, the LLVM infrastructure is not monolithic. The Frontend can be used to accept a wide range of General Purpose Programming Languages, the LLVM IR can be used to optimize the Intermediate Code

generated and the back-end is responsible for the retargetable code generator, i.e. LLVM Code Generator generates target specific machine code. This is ideal for a DSL that can accept varied inputs and be applied to various domains.

2.5.1 Domain in Focus : eBPF Verifier

The Domain we intend to focus on is eBPF. [3] The Extended Berkeley Packet Filter (eBPF) is a Linux subsystem that allows the safe execution of untrusted user-defined extensions inside the kernel. It relies on static analysis to protect the kernel against buggy and malicious extensions.[9] The verification of input programs passed to the the eBPF Verifier module, a solution that helps overcome the access limitations of a Linux kernel, allows a user to escalate user-space programs into kernel-space privilege.

However, the eBPF verifier, despite being continuously under development, still poses a number of restrictions to developers, such as

- False positives: many correct programs are rejected
- Lack of scalability: Cannot efficiently handle programs with many paths.
- Does not support loops and backward edges - compulsory loop unrolling has to be used.
- Lack of formal foundations, just DFG (Data Flow Graph) based verification
- Unclear or unfriendly error diagnostic messages.

2.5.2 Domain application of PRSafe

With the supportive and user-friendly features of PRFs, LLVM, PRSafe is eventually intended to be a pre-Verifier of sorts to the eBPF Verifier module. It offers the following features:

- Compensate for the lack of clear error diagnostics by the eBPF verifier module.
- Scalable
- Provides LLVM IR, a user-friendly, comprehensive Intermediate Representation that can be used to perform code optimization, retargetable code generation.
- Acceptance of finite loops such as for loops.

3. Design of PRSafe

PRSafe uses the ideas of Exanoke [7] to implement basic Primitive Recursive properties. This section describes the outline of PRsafe ' s design and construction. Some basic rules to tell if a functional program is primitive recursive:

- General recursion does not occur (Mutual recursion included)
- When recursion happens, it's always with arguments that are "strictly smaller" values than the arguments the function received.
- There is a base case argument that ensures the function always terminates.

3.1 EBNF Grammar

PRSafe proposes a simple grammar which facilitates basic operations and primitive recursion. The grammar is under modification to add new features and refine functionalities. However, the broad categorization of commands allowed in the language are : (i) Expressions (ii) Statements (iii) Functions composed of expressions and statements

- Expressions Includes Function calls.
- Statements Includes Variable declarations, Variable definition and assignment statements.
- Functions composed of expressions and statements Recursive functions are usually composed of other primitive recursive functions expressed using basic mathematical and logical operations such as addition, multiplication, comparison, etc.

  For example, in a factorial function,

```
int factorial (int n )
{
   if n==1
return 1;
   else if n > 1
return n * factorial (n-1);
}
```

the function is composed of multiplication, equality check and greater-than check . Even the recursive call is permitted to be made only with an argument (n-1) that is strictly lesser than (n).

3.1.1 Core Grammar

The core eBNF grammar of PRSafe is listed as follows.

$\langle program\rangle$ ::= '{'$\langle stmts\rangle$'}'

$\langle stmts\rangle$ ::= $\langle statement\rangle$
 | $\langle stmts\rangle$ $\langle statement\rangle$
 | $\langle empty\rangle$

A standard PRSafe program is composed of a block of statements, that may be empty or finite.

```
⟨statement⟩ ::=  ⟨variable Declaration⟩
  |  ⟨variable Definition⟩ | ⟨function_prototype⟩
  |  ⟨Function Declaration⟩                          |
     ⟨Function Definition⟩
  |  ⟨expr⟩
  |  'for'  ⟨ident⟩  '='  ⟨expr⟩  'to'  ⟨expr⟩  'do'
     ⟨statement⟩
  |  'PRINT' '('expression ')' ⟨expr⟩
  |  'RETURN' ⟨expr⟩

⟨block⟩ ::=  ⟨empty⟩
  |  '{'⟨stmts⟩ '}'
  |  '{' '}'

⟨variable Declaration⟩ ::=  ⟨ident⟩⟨ident⟩

⟨variable Definition⟩ ::=  ⟨ident⟩⟨ident⟩ '=' ⟨expr⟩

⟨function_prototype Defn⟩ ::=  ⟨ident⟩⟨ident⟩
     '('⟨function_prototype⟩')'⟨block⟩

⟨func_proto_args⟩ ::=  ⟨variable Declaration⟩
```

The statements accepted are similar to basic C++ blocks of a program, the types of statements are explained below.

- Declarative statements such as variable declarations, variable definitions, function prototypes, function definitions.
- Expression statements that comprise of all function calls, arithmetic expressions, variable definitions.
- Finite looping statement indicated by the 'for' keyword, followed by an expression.
- *Print* and *Return* statements.

```
⟨numeric⟩ ::=  ⟨Integer⟩
  |  ⟨Boolean⟩
```

To implement a Primitive Recursive Language, we require only natural numbers and conditionals to be expressed. Further, by restricting the data types that are allowed, we do not concern ourselves with type-safety issues at this point of development.
Hence, only the following two data types are accepted by the language currently.

- Integer
- Boolean

```
⟨expr⟩ ::=  ⟨ident⟩
  |  ⟨ident⟩ '=' ⟨expr⟩
  |  ⟨ident⟩ '(' ⟨call_args⟩ ')'
  |  ⟨numeric⟩
  |  ⟨expr⟩ ⟨bin_op⟩ ⟨expr⟩

⟨call_args⟩ ::=  ⟨expr⟩
  |  ⟨ident⟩⟨expr⟩
```

An expression in PRSafe basically evaluates to some result. It could be any of the following forms.

- A variable declaration or definition being declared or assigned to by a user-defined identifier.
- A function call defined by an arbitrary function name. It accepts as input a single argument of either of the following types - Integer, Boolean or another expression. Note: A function call could also be a recursive call, provided it only accepts recursive arguments that are "strictly smaller" than the original call argument. This is expressed by expressing the given recursive function call using a previously accepted primitive recursive function. The primitive recursive functions accepted are hence, strictly some composition of basic operations such as Addition function, Multiplication function, Successor Function, Zero Function.
- An arithmetic expression to be evaluated, comprising of any acceptable binary operator and acceptable operands.

```
⟨bin_op⟩ ::=  '=='
  |  '!='
  |  '<'
  |  '>'
  |  '+'
  |  '-'
  |  '*'
```

The set of operators allowed include the following standard operators :

- Assignment Operator '='
- Arithmetic Operators: '+' *(Addition)*, '-' *(Subtraction)*, '*' *(Multiplication)*,
- Comparison Operators: '<' *(Compare if less-than)*, '>' *(Compare if greater-than)*, '==' (Compare if equal to), '!=' (Compare if not equal to)

The grammar of PRSafe expresses that only finite loops, recursion with smaller arguments, Primitive Recursion functions, and composition of Primitive Recursive Functions can be accepted. However, in order to prove that only s̀trictly smallerárguments will be accepted, we plan to use an SMT solver

3.2 Code Generation performed by PRSafe and LLVM IR

Code Generation done by PRSafe has been implemented using the LLVM APIs. [10]. These APIs can generate IR Code that looks like human-readable assembly code, for any 'value' in a program. Any term that is represented in a source language, can be considered a 'value' in LLVM. Hence, LLVM can be used to generate code for operators, operands, functions, classes, any expression etc. There is no restriction on representation format. For example, in terms of bit width 64-bit Integer can be expressed as well as a 1-bit integer.

Sample High-Level Input:

```
void main (int i)
{
 i = 10
}
```

Corresponding PRSafe output generated:

```
$ ./toy testFunc
Succeeded to parse!
0x563e13337ed0
Creating variable declaration int i
Creating assignment for i
Creating integer: 10
Creating block
Creating function: main
Creating block
```

Corresponding LLVM IR generated:

```
define internal void @main.1(i64) {
entry:
  %i = alloca i64, addrspace(1)
   store i64 10, i64 addrspace(1) * %i
   ret void
}
```

4. Evaluation and Discussion

Table 1 shows a comparison between PRSafe and conventional langulages. In this section, we evaluate and discuss based on this table.

4.1 Mathematical Verification of PRSafe

Mathematical verification of PRSafe is currently a work-in-progress, just as additional functionalities are being added. We are currently using Z3, an easy-to-use SMT solver [11]. A Z3 script is a sequence of commands. For verification of the PRF properties of PRSafe, we specify a set of definitions as user-defined formulas.

For example, to restrict the properties of recursive functions in PRSafe, we can specify the following properties.
(1) All inputs and outputs can only be of Integer or Boolean Datatype
(2) Recursive calls are allowed in a function, only if the recursive call accepts arguments are ‘strictly smaller’ than the original argument. Hence, the recursive call must be expressed as a function expression that evaluates to a value lesser than the original function expression.

4.2 Alternate methods to PRSafe: F*

PRSafe acts as a verification language using mathematical functions and syntax specifications. However, instead of using PRFs to define a language, we can use languages such as F*, a general-purpose functional programming language with effects aimed at program verification. [12] Since F* harnesses the SMT solver Z3, it does not require a separate verification tool. F* can be used to perform the exact same functions such as ensuring type safety, proving termination, or designing a language based on a chosen mathematical model, such as Lambda Calculus. However, there is a certain learning curve involved in learning and verifying a programming language using F*.

At this stage in our work, we are still adding features to PRSafe and hence, we present purely a qualitative evaluation between F*, PRSafe and General purpose programming languages such as C.

5. Conclusion and Future Work

The main incentives to PRSafe is the user-friendly approach, the mathematical foundation in design, scope for optimization and scaling of features. PRSafe is able to reduce the burden on the user to design safe programs. Hence, it achieves its preliminary goals. However, the project has a long way to become a fully functional tool. Some of the work-in-progress includes the following ideas. Firstly, as PRSafe grows in functionalities, the PRF-based property checks must be completely verified using Z3. Second, a quantitative evaluation is required to compare benchmarks among methods equivalent to using PRSafe. Third, the unrolling of possible ‘safety-threatening’ features, such as mutual recursion in the space of eBPF can be explored.

Acknowledgement

This work is partially supported by enPiT (Education Network for Practical Information Technologies) at MEXT, Innovation Platform for Society 5.0 at MEXT, and Softbank Corporation.

References

[1] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*, 2004.

[2] Constant folding. `https://en.wikipedia.org/wiki/Constant_folding`.

[3] Jesper Dangaard Brouer. eBPF - extended Berkeley Packet Filter - Documentation. `https://prototype-kernel.readthedocs.io/en/latest/bpf/`. Copyright 2016, Revision 43e71113.

[4] Rajesh_Kr_Jha. Phases of a compiler. `https://www.geeksforgeeks.org/phases-of-a-compiler/`, 2020. Last Updated: 08-06-2020.

[5] Stefan Kahrs. Genetic programming with primitive recursion. pages 941–942, 01 2006.

[6] Douglas R Hofstadter. *Gödel, Escher, Bach: an eternal golden braid.* Penguin books. Basic Books, New York, NY, 1979.

[7] Chris Pressey. Exanoke, a functional language which is syntactically restricted to primitive recursive functions. `https://github.com/catseye/Exanoke`, 2013.

Table 1 Comparison with Conventional Languages

Tool / Metric of assessment	PRSafe	F*	C
Tool specific prerequisite knowledge	Negligible	Necessary	Necessary
Safety Properties	In-built syntactically	Needs specification	Lacks assurance
Platform independent Optimization	Allowed	Not Allowed	Not inherently allowed
User-friendly diagnostics	Available	subjective	subjective
Mathematical Foundations dependence	Yes	Yes	Not necessary
Feature rich - e.g. Finite Loop support, Type Safety (WIP)	Yes	Yes	Partially supported

[8] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda–a functional language with dependent types. In *International Conference on Theorem Proving in Higher Order Logics*, pages 73–78. Springer, 2009.

[9] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019.

[10] Llvm language apis. https://llvm.org/doxygen/.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS ' 08/ETAPS ' 08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[12] Inria Microsoft Research et al. F*, the general-purpose functional programming language aimed at program verification. https://github.com/FStarLang/FStar.