



HAL
open science

Verified programming and secure integration of operating system libraries in Coq

Shenghao Yuan

► **To cite this version:**

Shenghao Yuan. Verified programming and secure integration of operating system libraries in Coq. Systems and Control [cs.SY]. Université de Rennes, 2023. English. NNT : 2023URENS060 . tel-04405955

HAL Id: tel-04405955

<https://theses.hal.science/tel-04405955>

Submitted on 19 Jan 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601

*Mathématiques, Télécommunications, Informatique, Signal, Systèmes,
Électronique*
Spécialité : *Informatique*

Par

Shenghao Yuan

**Verified programming and secure integration
of operating system libraries in Coq**

Thèse présentée et soutenue à Rennes en salle Métivier, INRIA, le 8 décembre 2023 à 14h30

Rapporteurs avant soutenance :

Yongwang ZHAO Professeur à l'Université de Zhejiang, Chine
Gilles GRIMAUD Professeur des Universités à l'Université de Lille

Composition du Jury :

Présidente :	Sandrine Blazy	Professeur à l'Université de Rennes
Rapporteurs :	Yongwang ZHAO	Professeur à l'Université de Zhejiang, Chine
	Gilles GRIMAUD	Professeur des Universités à l'Université de Lille
Examineurs :	Emmanuel Bachelli	Chargé de Recherche, INRIA Rennes
	Frédéric Besson	Chargé de Recherche, INRIA Rennes
Dir. de thèse :	Jean-Pierre TALPIN	Directeur de Recherche, INRIA Rennes

Abstract

The C programming language presents a dual nature, offering low-level control over memory and high efficiency, while also being prone to error, especially in terms of memory management. Legacy C code remains in operation in critical fields such as finance, transportation, digital networks, and the Internet of Things (IoT). Despite its advantages, the potential for vulnerabilities, such as buffer overflows and use-after-free errors, presents significant challenges.

Formal methods offer a rigorous approach by abstracting low-level code into high-level models and mathematically verifying expected properties. This manuscript explores the use of Interactive Theorem Proving (ITP) within the proof assistant Coq to ensure program correctness.

The first part of this manuscript proposes an end-to-end verification approach that minimizes the verification gap between Gallina implementation and extracted C code. This approach starts with a proof model written in Gallina and certifies the expected properties of the model in Coq. It then derives a verified executable C implementation from this proof model.

The manuscript then applies this end-to-end approach to a subsystem of the IoT operating system RIOT-OS: rBPF. rBPF is a virtual machine integrated into RIOT-OS for sandboxing untrusted extensions. The current rBPF system, consisting of a verifier and a defensive interpreter, faces challenges related to safety and performance.

The second part of the manuscript presents a fault-isolating virtual machine, CertrBPF. CertrBPF is a formally verified model and implementation of rBPF in Coq. CertrBPF formalizes the syntax and semantics of all rBPF instructions, implements a formal model of the rBPF interpreter, completes the proof of fault isolation, and extracts C code from this formalization.

The third part of the manuscript introduces a JIT compiler to dynamically generate ARM binary code from rBPF bytecode. This JIT compiler is designed to speed up computation tasks in rBPF programs while reusing the existing rBPF interpreter CertrBPF to execute security-sensitive branch and memory instructions, subject to complex run-time checking.

Finally, we integrate CertrBPF and the JIT compiler into RIOT-OS and evaluate their performance on a popular ARM microcontroller. The results show significant improvements in security, memory footprint, and execution time compared to the existing rBPF interpreter.

RÉSUMÉ EN FRANÇAIS

Les fonctionnalités bas niveaux offertes par le langage de programmation C sont à la fois un atout et une source d'erreurs. D'une part, il a été conçu [KR02] pour offrir un contrôle très détaillé de la représentation des données au niveau octet en mémoire informatique. Cette fonctionnalité de bas niveau permet aux programmeurs de développer des artefacts hautement efficaces, économes en ressources, et faciles à lire par rapport au code d'assemblage au niveau machine. Par conséquent, des milliards de lignes de code C hérité sont toujours en service dans des domaines critiques (et non critiques), notamment la finance, les transports, les réseaux numériques et en particulier l'Internet des objets (IoT). D'autre part, le revers de la médaille est que l'utilisation du C facilite les erreurs de gestion de la mémoire de bas niveau, telles que le débordement de flottant, le dépassement de tableau, et l'utilisation après libération, etc. L'abstraction de bas niveau du code C accroît également les vulnérabilités dans des systèmes distribués complexes et parfois critiques, par exemple, en introduisant des erreurs logiques.

Il existe plusieurs façons de détecter les défauts logiciels. Une solution consiste à suivre un processus de développement logiciel strict et qualifiable pour mettre en œuvre des artefacts certifiés (*e.g.*, SCADE KCG [Ber07]) qui satisfont aux exigences de certification (*e.g.*, DO-178C [Rie17]) pour des domaines critiques spécifiques. Une autre approche consiste à concevoir un nouveau langage doté de fonctionnalités de sécurité spécifiques (*e.g.*, Rust [MK14] pour garantir la sécurité de la mémoire). Bien que ces mesures soient capables de détecter certaines vulnérabilités, elles ne peuvent généralement pas garantir l'absence de vulnérabilités, en particulier pas de manière rigoureuse sur le plan mathématique.

Techniques formelles. Les méthodes formelles offrent des garanties plus solides d'absence de bogues en abstrayant les programmes de bas niveau en modèles de haut niveau, permettant ainsi de raisonner mathématiquement sur leurs propriétés attendues. Par exemple, la ligne de production de code Airbus combine plusieurs méthodes formelles : i) L'analyseur statique Astrée [Cou+05] prouve l'absence de certains types spécifiques de bogues dans les programmes C, par exemple, les dépassements de tableau. ii) L'outil d'analyse formelle Frama-C [Kir+15] garantit les propriétés fonctionnelles des programmes C. iii) Le com-

pilateur vérifié [CompCert](#) [Ler09] traduit les programmes C en code binaire.

Les méthodes formelles englobent diverses techniques, notamment l’interprétation abstraite, la vérification de modèles, et la démonstration de théorèmes (automatique ou interactive). Parmi toutes les méthodes de vérification formelle, la preuve interactive de théorèmes (ITP) est probablement la technique la plus complète et rigoureuse pour garantir de manière constructive la correction d’un programme par le raisonnement mathématique sur l’exécution du programme (sémantique du programme). En résumé, le processus ITP implique : i) Une description mathématique des comportements du programme définie par l’humain, c’est-à-dire une sémantique formelle. ii) Des preuves mathématiques fournies par l’humain (dans certains langages de tactiques) de propriétés cibles. iii) La vérification de la correction des preuves par la machine. Toutes ces étapes sont réalisées au sein d’un *assistant de preuve*. Les assistants de preuve courants incluent [Coq](#) [BC13], [Isabelle/HOL](#) [NWP02], et [Lean](#) [Mou+15], etc. Par exemple, le compilateur CompCert est formellement vérifié en utilisant l’assistant de preuve Coq : toutes les langues intermédiaires et transformations pertinentes sont spécifiées dans le langage fonctionnel [Gallina](#) [Hue92], intégré dans Coq, et toutes les preuves vérifiées par machine concernant le théorème de préservation du comportement sont exprimées à l’aide du langage de tactiques [Ltac](#) [Del00] au sein de Coq.

Approches de vérification de bout en bout. Nous nous concentrons sur la spécification et la vérification des logiciels IoT, qui sont généralement écrits en C. Il existe des méthodes qui vérifient directement du code C de bas niveau écrit à la main, telles que VCC [Coh+09], VeriFast [JP08], RefinedC [Sam+21], et VST [App+14]. En revanche, cette thèse considère une approche de vérification de bout en bout différente qui modélise abstraitement des spécifications de haut niveau dans le langage de programmation d’un assistant de preuve. Ce modèle abstrait élimine les détails spécifiques du code de bas niveau, permettant aux utilisateurs de tirer pleinement parti de la puissance de l’assistant de preuve pour vérifier les programmes.

Les approches de vérification de bout en bout nécessitent également un compilateur vérifié ou de confiance pour relier les modèles abstraits aux implémentations de bas niveau. Des compilateurs existants tels que CertiCoq (Coq) [Ana+17], Cogent (Isabelle/HOL) [Riz+16], KaRaMeL (F*/Low*) [Pro+17], et d’autres ont été développés, mais aucun d’entre eux ne peut simultanément remplir les exigences suivantes :

- compilation vérifiée pour garantir que les propriétés attendues des spécifications de

- haut niveau sont préservées dans les programmes de bas niveau;
- hautes performances du code C généré, par exemple, exempt de gestion automatique de la mémoire (collecte des déchets);
- une base de calcul de confiance minimale (Trusted Computing Base - TCB).

L’objectif principal de cette thèse est de proposer une approche de vérification de bout en bout qui réponde à ces exigences. Nous appliquons cette approche pour vérifier un sous-système d’un système d’exploitation IoT du monde réel, [RIOT-OS \[Bac+18\]](#) : [rBPF \[ZB20\]](#).

rBPF. Dans de nombreux cas, les systèmes d’exploitation adoptent des techniques de confinement pour isoler les extensions non fiables dans le noyau. Pour ce faire, la communauté Linux a étendu les filtres Berkeley classiques [[MJ93](#)] (BPF) pour permettre l’exécution de code de machine virtuelle (VM) personnalisé en mode noyau. Cette VM est intégrée dans divers sous-systèmes, servant à des fins autres que le filtrage de paquets [[Fle17](#)]. L’extension de l’eBPF aux microcontrôleurs a conduit à la spécification du rBPF dans RIOT-OS.

Le rBPF actuel se compose d’un vérificateur compact et d’un interpréteur défensif sophistiqué, mais il est confronté à deux principaux défis :

- **Sécurité rBPF** : Tout comme le rôle de l’eBPF dans Linux, le rBPF fonctionne au sein du noyau sensible et critique de RIOT-OS. Étant donné qu’il fonctionne en mode noyau, le rBPF pourrait compromettre la sécurité et l’intégrité du système d’exploitation en exécutant du code non fiable de tiers. Le vérificateur rBPF pourrait ne pas parvenir à identifier des scripts malveillants ou erronés, et l’interpréteur pourrait ne pas être en mesure d’empêcher leur exécution. Cette menace est possible, compte tenu de plusieurs vulnérabilités de l’eBPF signalées à la communauté Linux à diverses occasions, telles que [CVE-2023-0160](#) (défaut de blocage), [CVE-2022-3646](#) (fuite de mémoire), [CVE-2022-3623](#) (attaque par condition de concurrence), et plus encore.
- **Performances rBPF** : Le rBPF sacrifie les performances en adoptant un interpréteur défensif comme moteur d’exécution. Bien que ce choix garantisse que du code non fiable, erroné ou malveillant puisse s’exécuter dans un environnement ouvert ou potentiellement hostile, tout en isolant les erreurs pour protéger l’intégrité de l’hôte, il conduit naturellement à des performances plus faibles par rapport à une implémentation Just-In-Time (JIT) qui optimise la vitesse d’exécution du programme

en traduisant un bytecode générique à partir d'un script source en un ensemble d'instructions spécifiques à la machine. Cependant, l'intégration d'un compilateur JIT Linux traditionnel dans le rBPF est difficile en raison de la complexité des stratégies de défense du rBPF. L'incorporation d'un compilateur JIT pourrait également introduire de nouvelles erreurs potentiellement plus insidieuses au niveau binaire, rendant plus difficile la garantie de la propriété d'isolement.

Contenu de cette thèse. Développer une machine virtuelle isolante de pannes pour les microcontrôleurs (MCU) pose deux défis majeurs. Le premier est d'intégrer la VM au sein du micro-noyau du MCU et de réduire la taille de son code et de son environnement d'exécution. L'autre est de réduire l'écart de vérification entre son modèle de preuve et le code réel en cours d'exécution.

Dans cette thèse, nous relevons ces défis et présentons la première vérification de bout en bout et la synthèse d'une machine virtuelle complète du monde réel pour la famille d'ensembles d'instructions BPF : CertrBPF, un interprète adapté aux architectures de MCU fonctionnant sous RIOT-OS, avec des ressources limitées.

Cette thèse expose nos solutions pour le développement et la preuve de la correction de rBPF. Plus précisément, nous présentons les contributions suivantes :

Approche de Vérification de Bout en Bout : Nous introduisons une approche de vérification de bout en bout pour réduire l'écart entre une implémentation Gallina vérifiée et son code C extrait non-vérifié. Notre point de départ est un modèle de preuve écrit en Gallina. Nous utilisons ce modèle de preuve pour certifier les propriétés attendues. À partir de ce modèle de preuve, nous dérivons ensuite un modèle de synthèse dont nous extrayons une version exécutable en Clight, que nous prouvons finalement pour effectuer les mêmes transitions d'état.

Interprète rBPF Certifié : CertrBPF est un modèle vérifié et une implémentation de rBPF en Coq. Nous formalisons la syntaxe et la sémantique de toutes les instructions rBPF, nous mettons en place un modèle formel de son interprète (femto-conteneur), nous achevons la preuve des propriétés critiques de notre modèle, et nous extrayons et vérifions le code CompCert C de cette formalisation. Notre approche de bout en bout obtient une machine virtuelle entièrement vérifiée. Non seulement la spécification Gallina de la VM est prouvée comme étant isolée du noyau et de la mémoire à l'aide de l'assistant de preuve Coq, mais l'interprétation directe de sa sémantique attendue sous forme de code CompCert C est elle-même vérifiée comme étant cor-

recte. Cela permet d’obtenir un programme binaire entièrement vérifié offrant une sécurité maximale et une empreinte mémoire minimale, et réduit la base de calcul de confiance (TCB) : CertrBPF, une machine virtuelle de niveau noyau efficace en mémoire qui isole les erreurs logicielles d’exécution à l’aide d’un code défensif et ne nécessite pas de vérification hors ligne.

Compilateur JIT rBPF Certifié : Nous introduisons JIT_{ALU} , une spécification basée sur Gallina d’un compilateur JIT qui génère dynamiquement du code binaire ARM à partir du bytecode rBPF. La compilation JIT accélère considérablement les tâches de calcul dans les programmes rBPF tout en réutilisant l’interprète rBPF CertrBPF existant pour exécuter les instructions de branches et de mémoire sensibles à la sécurité, soumises à une vérification complexe à l’exécution. L’implémentation exécutable en CompCert C de la spécification hybride VM+JIT, appelée HAVM, est directement obtenue à partir de sa spécification monadique en Gallina dans Coq. Nous fournissons également le code de liaison : une procédure `jit_call`, qui construit un environnement ARM approprié à partir de la VM pour sauter directement à l’adresse du code `jited`. Son implémentation se compose de 5 lignes de code en langage d’assemblage écrites en C, et sa spécification est formellement définie dans Coq. En combinant `jit_call` avec les implémentations C vérifiées de CertrBPF et JIT_{ALU} , nous obtenons une implémentation prête à l’emploi de HAVM.

Intégration Sécurisée et Évaluations : Nous intégrons CertrBPF et HAVM en tant que remplacement direct de l’interprète rBPF non vérifié actuel dans RIOT-OS. Nous évaluons ensuite de manière comparative les performances de CertrBPF intégré dans RIOT-OS, fonctionnant sur diverses architectures de microcontrôleurs 32 bits. Nos évaluations montrent qu’en pratique, CertrBPF apporte non seulement une sécurité, mais réduit également l’empreinte mémoire ainsi que le temps d’exécution. Comme prévu, nous observons des accélérations significatives par rapport aux interprètes rBPF existants. [Chapter 7](#) présente l’évaluation prometteuse des performances de notre HAVM par rapport à RIOT-OS CertrBPF et aux anciens micro-benchmarks Vanilla-VM sur une plateforme Cortex-M de développement.

ACKNOWLEDGEMENT

I would like to extend my heartfelt gratitude to my advisors, Jean-Pierre Talpin and Frederic Besson, for affording me the remarkable opportunity to working on operating systems-related and compilers-related formal verification using Coq. I have learned and gained significant insights from our weekly meetings and occasional discussions.

I also wish to express my profound appreciation to my PhD jury. It is an honor to share my research topic with such esteemed experts. I am deeply thankful to my reviewers, Yongwang Zhao and Gilles Grimaud, for dedicating their valuable time to review this document and offering invaluable comments.

My sincere thanks go out to my collaborators: Emmanuel Bachelli and his RIOT operating system, Samuel Hym and his innovative dx tool, and Koen Zandberg for his work on the rBPF virtual machine. I am also indebted to my colleagues: Benjamin Lion, Lucas Franceschino, Jean-Joseph Marty, and Stéphane Kastenbaum, for their discussions and many help.

My time in Rennes has been enriched by the daily activities and sports I enjoyed with friends, including badminton, tennis, ping-pong, and running. Lastly, I offer profound gratitude to my family for their unwavering support throughout my PhD journey.

TABLE OF CONTENTS

List of Figures	15
List of Tables	17
1 Introduction	19
1.1 Motivation	19
1.2 Contributions	22
1.3 Outline	23
2 State of the Art on End-to-End Verification Approaches and BPF Verification	25
2.1 End-to-End Verification Approaches	26
2.2 BPF verification	30
2.2.1 Conclusion	32
3 Background	33
3.1 Berkeley Packet Filters (BPFs)	33
3.1.1 cBPF vs eBPF	33
3.1.2 RIOT-OS rBPF	36
3.2 CompCert	37
3.2.1 CompCert Architecture	37
3.2.2 CompCert Programs	38
3.2.3 CompCert Memory Model	40
3.2.4 CompCert Simulation Framework	41
3.2.5 CompCert Ecosystem	45
3.3 ∂x Code Generator	46
3.4 Conclusion	51
4 An End-to-End Verification Approach in Coq	53
4.1 Discussion: Which Way Do We Select	53

4.2	A Workflow for End-to-End Verification in Coq	54
4.2.1	Proof-Oriented Specification	55
4.2.2	C-ready implementation	57
4.2.3	Translation Validation of C code	57
4.2.4	Summary	58
4.3	Applications	59
5	CertrBPF: A fully Verified rBPF Virtual Machine	61
5.1	A Proof-Oriented Virtual Machine Model	62
5.1.1	Syntax	62
5.1.2	Machine State	64
5.1.3	rBPF Interpreter	65
5.1.4	Proof of Isolation	69
5.2	A Synthesis-Oriented rBPF Interpreter	71
5.2.1	Synthesis Model	72
5.2.2	C-ready Model	77
5.3	Simulation Proof of the C rBPF Virtual Machine	79
5.4	CertrBPF Verifier	82
5.5	Optimization	84
5.5.1	check_mem Optimization	84
5.5.2	Equivalence Proof	86
5.6	Conclusion	95
6	CertrBPF-JIT	97
6.1	rBPF-32	98
6.2	Just-In-Time Compilation	101
6.2.1	Structure	102
6.2.2	Core Mapping	104
6.2.3	Interaction	106
6.3	Refinement of rBPF-32: rBPF-32-JIT	108
6.3.1	Symbolic CompCert ARM	108
6.3.2	Transition Semantics of rBPF-32-JIT	110
6.4	Hybrid JIT Interpreter	113
6.4.1	Overview	113
6.4.2	CompCert ARM Interpreter	115

6.4.3	Monadic JIT Compiler	116
6.4.4	Hybrid rBPF-32 Interpreter	117
6.4.5	rBPF-32 C Implementation	118
6.5	Discussion	120
6.5.1	Proof Overview	120
6.5.2	Defensive JIT _{ALU}	122
7	Evaluation	127
7.1	Implementation	127
7.1.1	Coq Implementation	127
7.1.2	C Implementation	128
7.2	Experiment	129
7.2.1	Experimental Evaluation Setup	129
7.2.2	Benchmarks	130
7.2.3	Research Question: Memory Footprint	130
7.2.4	Research Question: CertrBPF Interpreter Performance	131
7.2.5	Research Question: CertrBPF Interpreter Optimization	134
7.2.6	Research Question: HAVM Optimization	134
8	Conclusion	137
8.1	Summary	137
8.2	Perspectives	138
8.2.1	Short-term Perspectives	138
8.2.2	Long-term Perspectives	139
	Bibliography	141

LIST OF FIGURES

2.1	The F*/Low* workflow	26
2.2	The Cogent workflow	27
2.3	The CertiCoq workflow (solid arrow: verified, dashed arrow: proof underway)	28
2.4	The Cuf workflow	29
2.5	The JITK compilation	31
2.6	The <i>Jitk</i> compilation	32
3.1	Linux eBPF instruction encodings	34
3.2	Linux eBPF Workflow	36
3.3	RIOT-OS rBPF Structure	37
3.4	CompCert Architecture	38
3.5	Syntax of CompCert Programs	39
3.6	CompCert Backward Simulation	43
3.7	CompCert Forward Simulation	44
3.8	CompCert tools and related projects	46
3.9	∂x workflow	47
4.1	End-to-end verification and synthesis workflow	54
5.1	Chapter Structure: CertrBPF	61
5.2	Core syntax of rBPF instruction set	63
5.3	Maps between (C) physical memory and (CompCert) memory model.	65
5.4	Simulation relation R between st_{rbpf} , left, and rBPFClight, right.	80
5.5	Synthesis model: check_mem optimization	85
5.6	Function tree of the optimized interpreter	89
5.7	The simplification process.	90
6.1	Chapter Structure: CertrBPF-JIT	97
6.2	Syntax of rBPF-32 instruction set	99
6.3	JIT procedure: from rBPF-32 A1u32 binary to ARM binary	102

LIST OF FIGURES

6.4	state block layout (The first 52 byte)	103
6.5	HAVM Overview	113
6.6	Three cases of <i>entry point</i>	114
6.7	JIT Proof Overview	121
6.8	JIT Simulation Diagrams	121
6.9	JIT defensive procedure	123
6.10	JIT defensive procedure: shift instructions	124
6.11	JIT defensive procedure: <i>DIV</i> instruction	124
7.1	Time per instructions on the Cortex-M4 platform	132
7.2	Sliding window average on Cortex-M.	133
7.3	Execution time per instruction, on an Arm Cortex-M4 microcontroller	135

LIST OF TABLES

3.1	eBPF instruction classes	35
3.2	eBPF ALU64 instructions	35
5.1	Mapping relation in the ∂x -CompCert library	77
6.1	Mapping relation from rBPF-32 Alu32 reg to ARM	104
6.2	Mapping relation from rBPF-32 Alu32 imm to ARM	105
7.1	Coq code statistics of CertrBPF	127
7.2	C code statistics of CertrBPF	128
7.3	Memory footprint of rBPF engines	131
7.4	Execution time of real-world benchmarks (64-bit)	133
7.5	JIT compilation and execution time of alu32 instructions	136
7.6	Execution time of real-world benchmarks (32-bit)	136

INTRODUCTION

1.1 Motivation

The C programming language both benefits from and is burdened by its low-level idioms and features. On the one hand, it was designed [KR02] to provide fine-grained control over byte-level data representation in computer memory. This low-level feature allows programmers to develop highly efficient, resource-frugal, and easily readable artifacts in comparison to machine-level assembly code. Consequently, there are billions of lines of legacy C code¹ still in operation across critical (and non-critical) fields, including finance, transportation, digital networks, and especially the Internet of Things (IoT). At the same time, the flip side of the coin is that using C is error-prone because it makes low-level memory management mistakes easy, for instance, float overflow, array out-of-bounds, and use-after-free, etc. The low-level abstraction of C code also increases the vulnerabilities in complex and sometimes critical distributed systems, *e.g.*, logic errors.

There are several ways to detect defects in software. One solution is to follow a strict and qualifiable software development process (including testing) to implement certified artifacts (*e.g.*, SCADE KCG² [Ber07]) that satisfy certification requirements (*e.g.*, DO-178C [Rie17]) for specific critical fields. Another approach involves designing a new language with specific safety features (*e.g.*, Rust [MK14] for ensuring memory safety). While these measures are capable of detecting certain vulnerabilities, they usually cannot guarantee the absence of vulnerabilities, especially not in a mathematically rigorous manner.

Formal Techniques. Formal methods provide stronger guarantees for the absence of bugs by abstracting low-level programs into high-level models, allowing their expected properties to be mathematically reasoned about. For instance, the Airbus code production

1. Slogan: Code becomes legacy as soon as it's written.

2. SCADE stands for Safety-Critical Development Environment, KCG stands for 'qualified code generator'

line combines several formal methods: i) The static analyzer [Astrée](#) [Cou+05] proves the absence of some specific types of bugs in C programs, *e.g.*, array-out-of-bound. ii) The formal analysis tool [Frama-C](#) [Kir+15] guarantees the functional properties of C programs. iii) The verified [CompCert](#) [Ler09] compiler translates C programs into binary code.

Formal methods encompass various techniques, including abstract interpretation, model checking, and (automated or interactive) theorem proving. Among all formal verification methods, Interactive Theorem Proving (ITP) is arguably the most comprehensive and rigorous technique to constructively guarantee the correctness of a program through mathematical reasoning about program execution (program semantics). In essence, the ITP process involves: i) Human-defined mathematical description of program behaviors, *i.e.*, formal semantics. ii) Human-provided mathematical proofs (in some tactic languages) of target properties. iii) Machine-checked correctness of proofs. All of these steps are performed within a *proof assistant*. Common proof assistants include [Coq](#) [BC13], [Isabelle/HOL](#) [NWP02], and [Lean](#) [Mou+15], *etc.* For instance, the CompCert compiler is formally verified using the Coq proof assistant: all relevant languages and transformations are specified in the functional language [Gallina](#) [Hue92], embedded in Coq, and all machine-checked proofs regarding the correctness theorem of behavior preservation are expressed using the tactic language [Ltac](#) [Del00] within Coq.

End-to-end Verification Approaches. We focus on the specification and verification of IoT software, which is typically written in C. There are methods that directly verify hand-written low-level C code, such as [VCC](#) [Coh+09], [VeriFast](#) [JP08], [RefinedC](#) [Sam+21], and [VST](#) [App+14]. In contrast, this thesis considers a different end-to-end verification approach that abstractly models high-level specifications in the programming language of a proof assistant. This abstract model eliminates specific details from low-level code, allowing users to harness the full power of the proof assistant to verify programs.

End-to-end verification approaches also necessitate a verified or trusted compiler to connect abstract models with low-level implementations. Existing compilers like [CertiCoq](#) (Coq) [Ana+17], [Cogent](#) (Isabelle/HOL) [Riz+16], [KaRaMeL](#) (F*/Low*) [Pro+17], and others have been developed, but none can simultaneously fulfill the following requirements:

- Verified compilation to ensure that the expected properties of high-level specifications are preserved in the low-level programs.
- High-performance generated C code, *e.g.*, free of automatic memory management (garbage collection).

- A minimal trusted computing base (TCB).

The primary objective of this thesis is to propose an end-to-end verification approach that meets these requirements. We apply this approach to verify a subsystem of a real-world IoT operating system (OS), [RIOT-OS \[Bac+18\]](#): rBPF [[ZB20](#)].

rBPF. In many cases, operating systems adopt sandboxing techniques to isolate untrusted extensions within the kernel. To achieve this, the Linux community extended the classical Berkeley Packet Filters [[MJ93](#)] (BPF) to enable the execution of custom in-kernel virtual machine (VM) code. This VM is integrated into various subsystems, serving purposes beyond packet filtering [[Fle17](#)]. The extension of eBPF to micro-controllers led to the specification of rBPF in RIOT-OS.

The current rBPF consists of a compact verifier and a sophisticated defensive interpreter, but it faces two main challenges:

- **rBPF Security:** Similar to eBPF’s role in Linux, rBPF operates within the sensitive and critical RIOT-OS kernel. As it functions in-kernel, rBPF could potentially compromise the operating system’s security by executing untrusted code from third parties. The rBPF verifier might fail to identify malicious or erroneous scripts, and the interpreter might be unable to prevent their execution. This threat is possible, considering multiple eBPF vulnerabilities reported to the Linux community on various occasions, such as [CVE-2023-0160](#) (deadlock flaw), [CVE-2022-3646](#) (memory leak), [CVE-2022-3623](#) (race condition attack), and more.
- **rBPF Performance:** rBPF sacrifices performance by adopting a defensive interpreter as its execution engine. While this choice ensures that untrusted, erroneous, or adversarial code can run in an open or possibly hostile environment, while isolating faults to protect the host’s integrity, it naturally leads to lower performance compared to a Just-In-Time (JIT) implementation that optimizes program execution speed by translating generic bytecode from a source script into the machine-specific instruction set. However, integrating a traditional Linux JIT compiler into rBPF is challenging due to the complexity of rBPF’s defensive strategies. Incorporating a JIT compiler could also introduce new and potentially more insidious errors at the binary level, making it harder to ensure the isolation property.

1.2 Contributions

Developing a fault-isolating virtual machine for Microcontrollers (MCUs) poses two major challenges. One is to embed the VM within the MCU’s micro-kernel and minimize its code size and execution environment. Another is to reduce the verification gap between its proof model and the actual running code.

In this thesis, we address these challenges by presenting the first end-to-end verification workflow, synthesizing a comprehensive real-world virtual machine for the BPF instruction set family: CertrBPF, and designing a JIT compiler to accelerate CertrBPF.

This thesis outlines our solutions for developing and proving the correctness of rBPF. Specifically, we present the following contributions:

End-to-end Verification Approach: We introduce an end-to-end verification approach to remove the gap between a verified Gallina implementation and its unverified extracted C code. Our starting point is a proof model written in Gallina. We use this proof model to certify expected properties. From this proof model, we then derive a synthesis model of which we extract an executable version in Clight, that we finally prove to perform the same state transitions.

Certified rBPF Interpreter: CertrBPF is a verified model and implementation of rBPF in Coq. We formalize the syntax and semantics of all rBPF instructions, implement a formal model of its interpreter (femto-container), complete the proof of critical properties of our model, and extract and verify CompCert C code from this formalization. Our end-to-end approach obtains a fully verified virtual machine. Not only is the Gallina specification of the VM proved kernel- and memory-isolated using the Coq proof assistant, but the direct interpretation of its intended semantics as CompCert C code is, itself, verified correct. This yields a fully verified binary program of maximum security and minimal memory footprint and reduced the TCB: CertrBPF, a memory-efficient kernel-level virtual machine that isolates runtime software faults using defensive code and does not necessitate offline verification.

Certified rBPF JIT Compiler: We introduce JIT_{ALU} , a Gallina-based specification of a JIT compiler that dynamically generates ARM binary code from rBPF byte-code. This numerical accelerator significantly speeds up computation tasks in rBPF programs while reusing the existing rBPF interpreter CertrBPF to execute security-sensitive branch and memory instructions, subject to complex run-time checking. The executable CompCert C implementation of the hybrid VM+JIT specification,


named hybridly accelerated virtual machine (HAVM for short), is directly obtained from its monadic Gallina specification in Coq. We also provide the glue code: a `jit_call` procedure, that builds a proper ARM environment from the VM to directly jump at the address of the `jited` code. Its implementation consists of 5 assembly code lines written in C and its specification is formally defined in Coq. By combining `jit_call` with the verified C implementations of CertrBPF and `JITALU`, we obtain a readily executable implementation of HAVM.

Secure Integration and Benchmarks: We integrate CertrBPF and HAVM as a drop-in replacement of the current, non-verified rBPF interpreter in RIOT-OS. We then comparatively evaluate the performance of CertrBPF integrated in RIOT-OS, running on various 32-bit micro-controller architectures. Our benchmarks demonstrate that, in practice, CertrBPF not just gains security, but reduces memory footprint as well as execution time. As expected, we observe significant speedups compared to existing rBPF interpreters. [Chapter 7](#) reports the promising performance evaluation of our HAVM against RIOT-OS CertrBPF and earlier Vanilla-VM micro-benchmarks on a development Cortex-M platform.

1.3 Outline

The main body of this thesis is structured as follows: [Chapter 2](#) introduces the state of the art, including end-to-end verification and BPF verification. [Chapter 3](#) presents the BPF family, CompCert, and the ∂x code extraction tool. [Chapter 4](#) presents our end-to-end verification approach, which involves formally refining monadic Gallina programs into C programs. This methodology consists of a proof model with certified properties, a refined synthesis model with optimization, and a verified Clight model with behavior-equivalent proof. [Chapter 5](#) defines CertrBPF, a verified model and implementation of the rBPF virtual machine in Coq. In particular, its proof model encompasses the semantics of our VM and isolation theorems, and the synthesis and implementation models are proven to satisfy the refinement relation, and a formally verified verifier establishes the invariants needed by the VM. In [Chapter 6](#), we present `JITALU`, a JIT compiler dynamically translating rBPF ALU32 bytecode to ARM binary code. This numerical accelerator is designed to speed up computation tasks. We then detail our solution to integrate `JITALU` into CertrBPF to create a hybrid interpreter. In this design, ALU32 bytecode is compiled by `JITALU` and executed directly by the hardware, while the more complex bytecode

(defensive memory operations) is still interpreted by CertrBPF. [Chapter 7](#) conducts a case study on the performance of our two generated VM implementations, CertrBPF and HAVM, in relation to off-the-shelf RIOT femto-containers. [Chapter 8](#) concludes and presents avenues for future work.

The CertrBPF project is the main contribution of the work package 3 in the Inria challenge project [RIOT-fp](#), which aims to provide formal proofs on RIOT-OS components. The source code associated with this thesis, and more broadly, with the entire CertrBPF project, is publicly available online in the project's GitLab repository [[Cer23a](#); [Cer23b](#)]. We provide links to this online source code for several definitions and theorems in this thesis in the form of Coq logos .

STATE OF THE ART ON END-TO-END VERIFICATION APPROACHES AND BPF VERIFICATION

Since Tony Hoare proposed the manifesto ‘The Verified Software Initiative’ that aims to construct error-free software systems, the successful verification of realistic software components has been established, especially for compilers and operating systems.

- In terms of compilers, the CompCert project [Ler09], led by Xavier Leroy, successfully verifies a C compiler that guarantees the compilation correctness from a large C99 subset (CompCert Clight) to various target assembly languages. Further details of CompCert are introduced in Section 3.2. Subsequently, many verified compilers are proposed: CakeML [MO12] implements a verified compilation from a subset of Standard ML to assembly, the Vellvm project [Zha+13] focuses on building a verified LLVM compiler, Vélus [Bou+17] is a verified compiler from the Lustre dataflow synchronous language [Cas+87] to CompCert Clight, and so on.
- In terms of verified OS kernels, the seL4 project [Kle+09] is the first to build a proof of functional correctness for a realistic microkernel, the proof is conducted over a high-level specification and then propagated down to a concrete implementation. Ironclad [Haw+14] establishes end-to-end security properties from the application layer down to kernel assembly. It uses the Dafny verifier [Lei10], built on the Z3 SMT solver [DB08], to help automate proofs. Other verified OS kernels are implemented for various purposes, *e.g.*, CertiKOS with multicore support [Gu+16], $\mu\text{C}/\text{OS-II}$ [Xu+16] for interrupt reasoning, Hyperkernel [Nel+17] with a high degree of proof automation, verified Zephyr RTOS [ZS19] with concurrent buddy memory allocation, and the Pip proto-kernel [Jom+18a; Jom+18b] for memory isolation on memory management unit (MMU).

In this chapter, we narrow our focus to two specific topics:

- End-to-end verification approach ($X\text{-to-}C$): This part belongs to the topic of compiler verification, it involves a proof-assistant-related language on one side and the C programming language on the other.
- BPF-related verification: This topic involves the verified OS components, and it covers the verification of the Linux eBPF verifier, interpreter, and JIT compiler.

2.1 End-to-End Verification Approaches

F* [Swa+13] is a proof-oriented functional language with effects and has been used to develop many high-assurance cryptographic algorithms, such as Chacha20, Poly1305, and SHA-3. For low-level code verification, F* embeds a domain-specific language (DSL) called Low* [Pro+17], which is a subset of F*. Low* includes a lower-level C-like memory model and libraries of C-style arrays and structs. As shown in Figure 2.1, F* also offers a compiler from Low* to C named KaRaMeL. While KaRaMeL has an on-paper semantics-preservation proof from the Low* semantics model λow to CompCert Clight [BL09], its current implementation, written in the functional language OCaml, remains unverified.

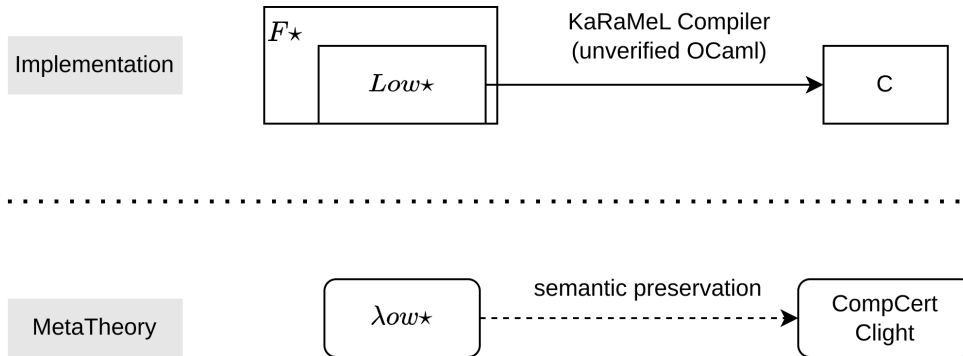


Figure 2.1 – The F*/Low* workflow

Comparison.

- *pros*: The F*/Low* approach offers a higher level of automation compared to our methodology. KaRaMeL incorporates a built-in mapping relation from F*/Low* types to C types whereas our methodology additionally requires users to manually rename their synthesis model with ∂x type configurations.

- *cons*: The F*/Low* approach involves a larger TCB which includes the F* type-checker, the Z3 SMT solver, and the KaRaMeL compiler. In contrast, our intended TCB is only limited solely to the Coq type-checker.

Another approach is Cogent [Riz+16], which aims to develop verified applications on top of the SeL4 [Kle+09] micro-kernel. Cogent, as depicted in Figure 2.2, consists of a functional language with linear types for specifying source programs and generates C code along with Isabelle/HOL proof information. Cogent lacks built-in recursion, and its iteration is expressed through integrated foreign function interfaces (FFI) that are verified by users. The Cogent compiler offers certifications to verify that the extracted C code refines a high-level Isabelle/HOL functional specification within the Isabelle/HOL proof assistant. Users can then prove that the Isabelle/HOL specification preserves the expected correctness properties.

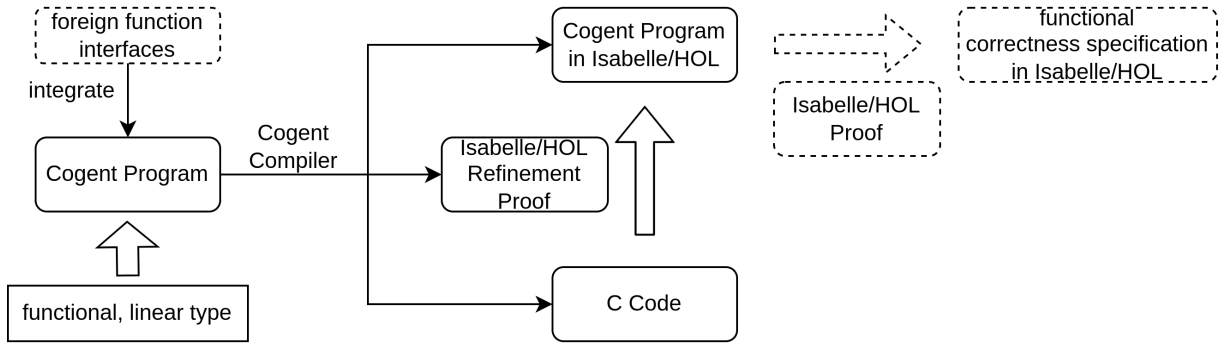


Figure 2.2 – The Cogent workflow

Comparison.

- *pros*: Cogent offers a higher degree of proof automation in contrast to our methodology. It provides a certifying compiler that ensures the simulation relation between the generated Isabelle/HOL model and extracted C code, In our approach, we use an unverified code generator, which needs an additional simulation proof from each input Gallina model to the output C implementation.
- *cons*: Our method is more direct than the co-specification approach (Cogent + Isabelle/HOL) in Cogent: We directly formalize specifications in Gallina that is embedded in Coq, then translate Gallina specifications into C code and performs the end-to-end verification in Coq.

When the scope is narrowed down to the topic of converting Gallina programs into executables, there are various techniques.

To begin with, Coq comes with a built-in extraction mechanism [Let02] that generates OCaml, Haskell, or Scheme. This path has a rather large TCB (Coq extraction and a compiler). CertiCoq [Ana+17] is an ongoing project aiming at generating CompCert C code from Gallina. This is achieved through several specific Intermediate Representations (IRs) and multiple transformation passes, as shown in Figure 2.3. The project’s initial two passes, named reification, and erasure, are part of the MetaCoq project [Soz+20]. After the eta expansion of constructors and patterns, as well as the let-binding of environment, CertiCoq introduces two alternative transformations: the Administrative Normal Form (ANF) or the Continuation-Passing Style (CPS) to generate the λ_{ANF} intermediate representation. ANF and CPS are two common low-level functional intermediate representations. Importantly, the λ_{ANF} representation in CertiCoq is syntactically a superset of CPS, and this allows users to obtain a λ_{ANF} program by choosing either ANF or CPS conversion to λ_{ANF} . Then the λ_{ANF} is compiled to λ_{ANF^C} that is a subset of the ANF language without nested functions. The final step involves C code generation including two procedures: one for handling CPS code and another for the full λ_{ANF^C} representation.

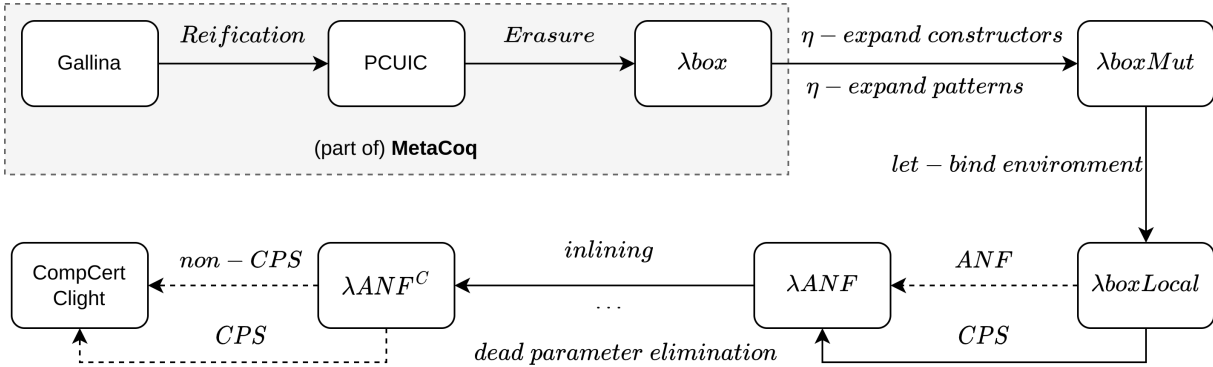


Figure 2.3 – The CertiCoq workflow (solid arrow: verified, dashed arrow: proof underway)

Comparison.

- *pros*: Similar to CakeML [MO12], CertiCoq performs a compilation from the full functional language Gallina to CompCert C, while our approach is limited to a subset of Gallina in monadic form. Once CertiCoq is completed, it will allow one to rely on a small TCB, akin to ours.
- *cons*:
 - *Readability*: CertiCoq often yields C code that is less readable compared to our approach, which produces C programs much more conducive to manual review.

- *Garbage Collection*: As a generic functional language compiler, CertiCoq generates C code with an (verified) external garbage collector, which is usually unsuitable for real-time IoT operating systems, *e.g.*, RIOT-OS. In contrast, our method is free of garbage collection.

Æuf [Mul+18] is another tool for compiling Gallina to C. The first compilation step, as shown in Figure 2.4, involves reflecting Gallina terms into corresponding Abstract Syntax Trees (ASTs) within the Æuf source language. This Æuf language is a lambda-lifted simply typed lambda calculus over a particular set of base types. The reflection procedure does not require to be trusted, as it is verified using translation validation [PSS98]. Subsequently, The process translates Æuf ASTs into the CompCert Cminor IR, followed by the reuse the CompCert backend to obtain the target machine code.

Comparison.

- *pros*: Æuf enjoys a higher degree of proof automation compared to our methodology.
- *cons*: Similar to CertiCoq, the C code generated by Æuf also relies on an unverified garbage collector. This can often result in increased overhead or memory footprint, which are highly constrained resources in IoT devices.

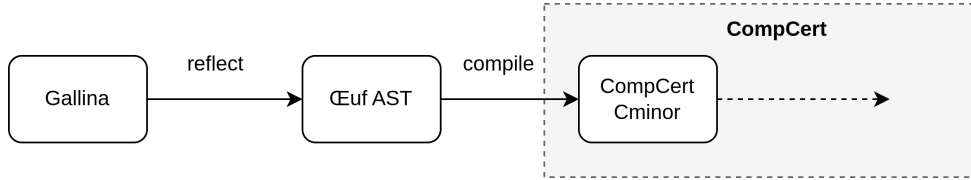


Figure 2.4 – The Æuf workflow

Codegen [Tan21] converts Gallina to C with partial evaluation. It primarily focuses on eliminating polymorphism and dependent types from Gallina, transforming Gallina functions into a form that closely resembles C within the Gallina language, and subsequently extracting C code from it.

Comparison.

- *pros*: Codegen offers the ability to configure the representation of values. For example, it can safely replace natural numbers with finite-size integers.
- *cons*: The main limitation of Codegen is that its transformations are *unverified*.

Rupicola [Pit+22] presents an original and promising approach that regards a compiler as a partial decision procedure. It comprises a proof search procedure, which may either

fail or produce a target program in bedrock2 [Erb+21] (a C-like low-level language AST embedded in Coq) along with a proof of equivalence.

Comparison.

- *pros*: Rupicola offers a higher level of proof automation compared to our approach.
- *cons*: Currently, Rupicola has only been tested for small algorithms, while our methodology aims for end-to-end verification of complex real-world applications.

2.2 BPF verification

Several existing formal approaches have addressed the Linux eBPF verifier, including verification of the soundness of range analysis in the verifier [SH23; Vis+23], abstract interpretation (PREVAIL [Ger+19]), and symbolic evaluation (Serval [Nel+19]).

Comparison.

- *pros*: All of the above approaches are designed for a sophisticated verifier, whereas our target, RIOT-OS rBPF, employs a tiny verifier.
- *cons*: None of these approaches can guarantee the termination of eBPF programs. In contrast, our scenario naturally guarantees termination, as rBPF introduces a default fuel mechanism for termination.

The JITK framework [Wan+14] uses Coq to implement and verify the correctness of a JIT compiler for the classic Berkeley Packet Filter language (not eBPF) in the Linux kernel. As depicted in Figure 2.5, JITK first introduces a high-level specification language called System Call Policy Language (SCPL) to specify the desired system call policies, It then implements a verified compiler for translating SCPL rules to BPF bytecode. Subsequently, JITK translates the BPF bytecode into the CompCert Cminor intermediate representation and leverages the CompCert backend to generate target code. The JITK compiler is extracted to OCaml implementation using the Coq extraction mechanism.

Comparison.

- *pros*: JITK supports the specification of system call policies as BPF filters. In contrast, our CertrBPF only supports limited build-in RIOT-OS system calls as same as the vanilla rBPF.
- *cons*: JITK adopts the Coq extraction mechanism to translate their JIT into an executable OCaml implementation, which runs in the Linux user space and requires modifications to the kernel for upcalls. Additionally, the Coq extraction approach is

not suitable for RIOT-OS, a resource-limited IoT operating system, due to JITK’s dependency on the OCaml runtime, an assembler, and a linker. Conversely, our method targets executable implementations written in CompCert C.

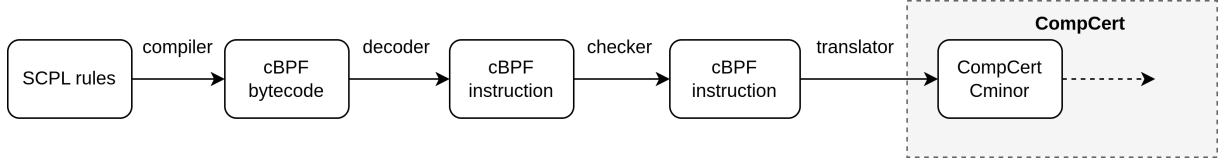


Figure 2.5 – The JITK compilation

JITSYNTH [Gef+20] is a tool designed for synthesizing verified JITs for in-kernel DSLs. As shown in Figure 2.6, it takes input information, including the syntax, semantics, and abstract register machine state for both the source and target instruction set architectures (ISA). The semantics are expressed in the form of an interpreter, and the machine state contains a register map, a memory model, and a program counter. By establishing mapping relations from the source state to the target state, JITSYNTH synthesizes a *mini* compiler from every instruction in the source language to a sequence of instructions in the target language. These per-instruction compilers are then composed into a full compiler using a trusted outer loop and a switch statement. JITSYNTH has been applied to synthesize a JIT compiler from eBPF to RISC-V.

Comparison.

- *pros*: JITSYNTH offers a method for synthesizing verified JITs from interpreters, while our case study implements a verified rBPF interpreter and an unverified JIT compiler respectively.
- *cons*: JITSYNTH exhibits a verification gap between their formal models and the low-level C implementation, as the final C artifact is manually written. We bridge this verification gap by leveraging our end-to-end verification approach in Coq to generate verified C implementations.

JITTERBUG [Nel+20] is a framework to write in-kernel JITs and prove them correct. This framework consists of three key components: i) formalizing the specification for JIT correctness, *i.e.*, the behavioral equivalence between the abstract machines of eBPF and target architectures, ii) proposing an automated proof strategy using symbolic evaluation to prove JIT correctness and iii) defining a C DSL for developing JITs, the DSL is a shallow embedding of a structured subset of C in Rosette [TB14], a solver-aided host

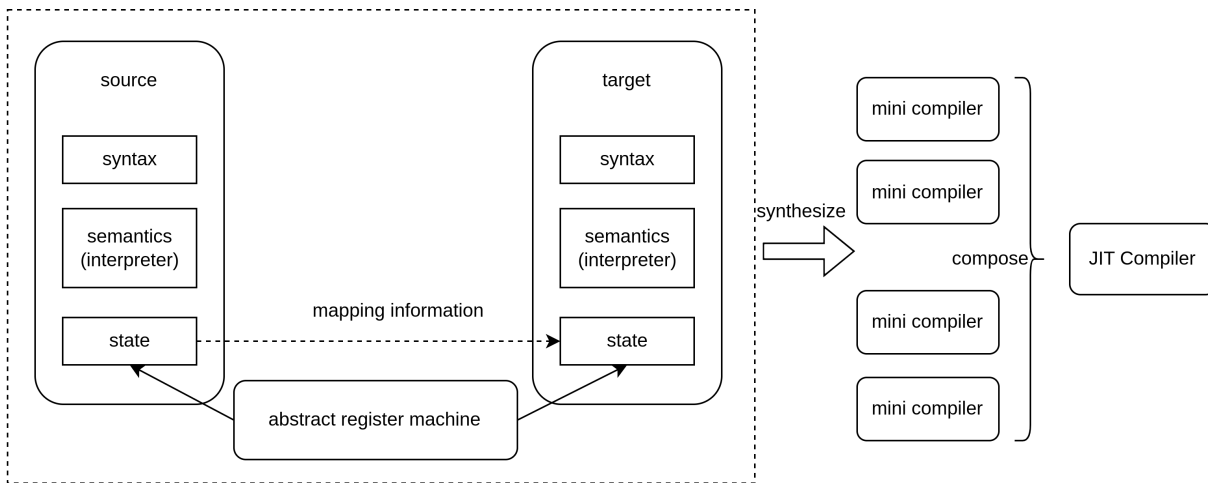


Figure 2.6 – The *Jitk* compilation

language. JITTERBUG has been used to implement and verify a new eBPF JIT for 32-bit RISC-V architectures.

Comparison.

- *pros*: JITTERBUG demonstrates a high degree of proof automation by using Rosette to formalize JITs specifications. While our approach requires more manual proof work.
- *cons*: Similar to JITSYNTH, JITTERBUG faces a verification gap, wherein the low-level C implementation is extracted through an unverified extraction mechanism. In contrast, Our intended goals are verified C programs.

2.2.1 Conclusion

The verification of realistic compilers and operating systems have been the subjects of vast development and verification efforts due to the sheer code size of the artifacts at stake. These full-scale case studies gave rise to new strategies and methodologies to address the challenge of verifying large-scale software. This chapter introduces the related work about the end-to-end verification approach and BPF verification, and shows the comparison between each related research and our methodology.

BACKGROUND

In this chapter, we introduce essential concepts necessary for our refinement methodology along with a verified rBPF implementation. We begin by focusing on BPF and its variants, with particular attention to Linux eBPF and RIOT-OS rBPF. Following that, we provide an overview of the formally verified CompCert C Compiler, including its intermediate languages, memory model, and the simulation framework. Finally, we discuss the key features of the ∂x code generation tool, which offers automatic but unverified extraction from monadic Gallina specifications to executable C code.

3.1 Berkeley Packet Filters (BPFs)

3.1.1 cBPF vs eBPF

Originally, Berkeley Packet Filters (BPF) [MJ93] was designed to provide Unix-BSD systems with network packets filtering capabilities and tools such as `tcpdump`. This classical BPF, also known as cBPF, takes the form of an assembly language that defines a virtual RISC-like ISA in which succinct and cautiously written scripts can be executed to parameterize privileged, mission-critical, network stacks. cBPF is highly restrictive and limited, featuring only two registers and bytecode interpretation. This restrictiveness becomes an obstacle for emerging scenarios that require rich functionality and low overhead.

For machines like PCs, servers, and routers, the Linux community extended the concept of cBPF to provide ways to run custom in-kernel virtualized (VM) code, hooked as "plugins" to various services and for varieties of purposes beyond packet filtering [Fle17]. This expanded version of BPF, known as eBPF or Linux eBPF, includes:

- *ISA*: Derived from the 64-bit RISC-V family, it offers 10 general-purpose registers as well as a read-only frame pointer register.
- *verifier*: This component statically analyzes eBPF binary instructions, rejecting all potentially unsafe programs. The [C implementation](#) spans over 10 thousands of lines

of code (KLOC).

- *interpreter*: It executes eBPF binary instruction one by one.
- *just-in-time (JIT) compiler*: This module translates eBPF binary into various 64/32-bit architectures, such as x86, ARM, and RISC-V.

eBPF ISA encoding. The eBPF ISA has two instruction encodings: little-endian encoding and big-endian encoding, as shown in [Figure 3.1](#).

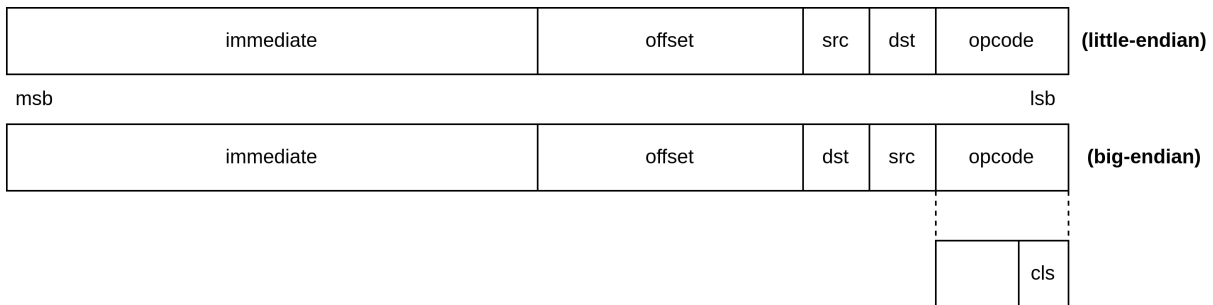


Figure 3.1 – Linux eBPF instruction encodings

A 64-bit eBPF instruction, from least significant bit (lsb) to most significant bit (msb), consists of the following fields:

- 8-bit opcode
- 4-bit destination register and 4-bit source register (their locations are target-dependent)
- 16-bit signed integer offset
- 32-bit signed integer immediate value

Most instructions do not use all of these fields, so eBPF specifies that unused fields should be all-zero. Details of eBPF instruction encodings can be found in the [eBPF Instruction Set Specification](#).

eBPF uses the three least significant bits of the opcode to define instruction classes (`cls`), as shown in [Table 3.1](#). The instruction class includes memory load (`0x00` and `0x01`), memory store (`0x02` and `0x03`), 32-bit and 64-bit arithmetic instruction (`0x04` and `0x07`), and branch (`0x05` and `0x06`).

Consider the 64-bit arithmetic instruction class (ie `0x07`) shown in [Table 3.2](#), both opcodes `0xX7` and `0xXf` (X representing an arbitrary hexadecimal number) are in this class. eBPF specifies such as `0x07` representing the 64-bit addition with immediate value instruction, `0x0f` denoting the 64-bit addition with source register instruction, and `0x17`

Table 3.1 – eBPF instruction classes

Class	Value	Description
BPF_LD	0x00	non-standard load operations
BPF_LDX	0x01	load into register operations
BPF_ST	0x02	store from immediate operations
BPF_STX	0x03	store from register operations
BPF_ALU32	0x04	32-bit arithmetic operations
BPF_JMP64	0x05	64-bit jump operations
BPF_JMP32	0x06	32-bit jump operations
BPF_ALU64	0x07	64-bit arithmetic operations

for the 64-bit subtract with immediate value instruction, etc. For example, assuming a little-end target, the 64-bit binary ‘0x000000000000250f’ corresponds to ‘BPF_ADD64 R5 R2’ which means adding the value of destination register R5 to the value of source register R2 and updating R5 with the result.

Table 3.2 – eBPF ALU64 instructions

Instructions	Opcode	Description
BPF_ADD64 dst imm	0x07	addition with immediate value
BPF_ADD64 dst src	0x0f	addition with register
BPF_SUB64 dst imm	0x17	subtract with immediate value
...

eBPF verifier. The eBPF validation process involves static analysis of eBPF bytecode before execution. It establishes two key properties for eBPF bytecode: *memory safety* and *termination*. For *memory safety*, the eBPF verifier ensures that the program only accesses memory locations within its allocated regions. To validate eBPF program *termination*, the verifier explores all reachable execution paths to ensure they safely terminate without errors. The eBPF verifier rejects programs beyond a [finite complexity](#) threshold of one million instructions.

eBPF workflow. While the Linux kernel expects eBPF programs to be loaded in the form of bytecode, application developers often prefer to write these programs in a high-level abstraction, *e.g.*, pseudo-C code. Then, these programs are translated into bytecode

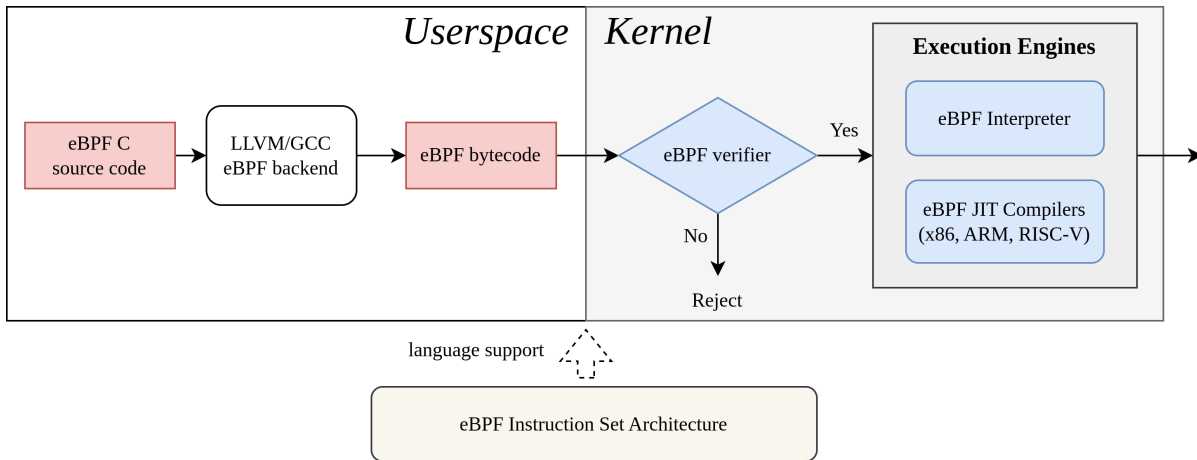


Figure 3.2 – Linux eBPF Workflow

using compiler suites like LLVM or GCC. As shown in Figure 3.2, if the eBPF verifier validates the provided bytecode scripts, the selected eBPF execution engine runs them.

Linux eBPF has broad extensions for specific purposes, For instance, a userspace eBPF (**uBPF**) enables the execution of eBPF programs on non-Linux systems, and Microsoft’s [ebpf-for-windows](#) builds on uBPF and the PREVAIL formal verifier.

3.1.2 RIOT-OS rBPF

eBPF was then ported to micro-controllers, resulting in the RIOT-OS rBPF specification. Just like eBPF, rBPF is designed as a 64-bit register-based VM, using fixed-size 64-bit instructions and a reduced instruction set architecture derived from eBPF. rBPF also employs a fixed-size stack (512 bytes) and excludes any heap interaction, thereby minimizing VM memory overhead in RAM.

rBPF comprises two key components: a compact [verifier](#) (less than 0.1 KLOC) for basic validation, *e.g.*, excluding illegal opcodes, and a defensive interpreter for execution.

The primary distinction between rBPF and eBPF is that rBPF adopts dynamic run-time checking to implement its fault-isolated, defensive strategies, while eBPF employs static analysis techniques via an offline, sophisticated verifier. Users of rBPF must declare all memory regions accessed by their programs, specifying fine-grained permissions for each region. The memory region declaration is defined as a C linear structure, encompassing details like start address pointer, size, and permission type.

```

struct mem_region {
    mem_region *next;      // pointing to next memory region
  
```

```

const uint8_t *start; // starting address of current region
size_t len;           // offset
uint8_t flag;        // permission type
};

```

The workflow of rBPF mirrors that of eBPF, as shown in Figure 3.3. The main difference is that rBPF only leverages an interpreter to execute the bytecode script.

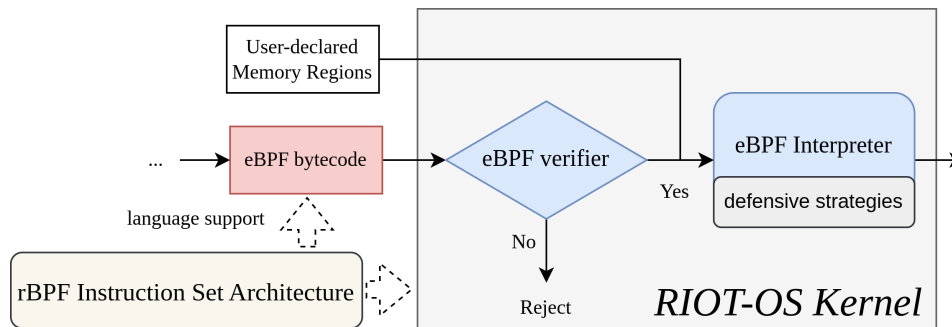


Figure 3.3 – RIOT-OS rBPF Structure

3.2 CompCert

CompCert [Ler09] is a C compiler that has been both programmed and proved correct using the Coq proof assistant. It compiles C programs into assembly programs that support mainstream target architectures (32 or 64-bits), including x86, ARM, and RISC-V. CompCert follows a *software-proof* codesign approach where development is carried out directly in Coq alongside its proof. It uses Coq’s extraction facility to generate executable OCaml code capable of compiling C programs.

This section mainly describes the architecture, programs, memory model, and simulation theorem of the CompCert compiler, which are essential for our work.

3.2.1 CompCert Architecture

As depicted in Figure 3.4, the CompCert C compiler is structured as a pipeline of 20 compilation passes that bridge the gap between source C code and target object files. It traverses 11 intermediate languages (IRs) in the process. The passes can be grouped into three successive phases:

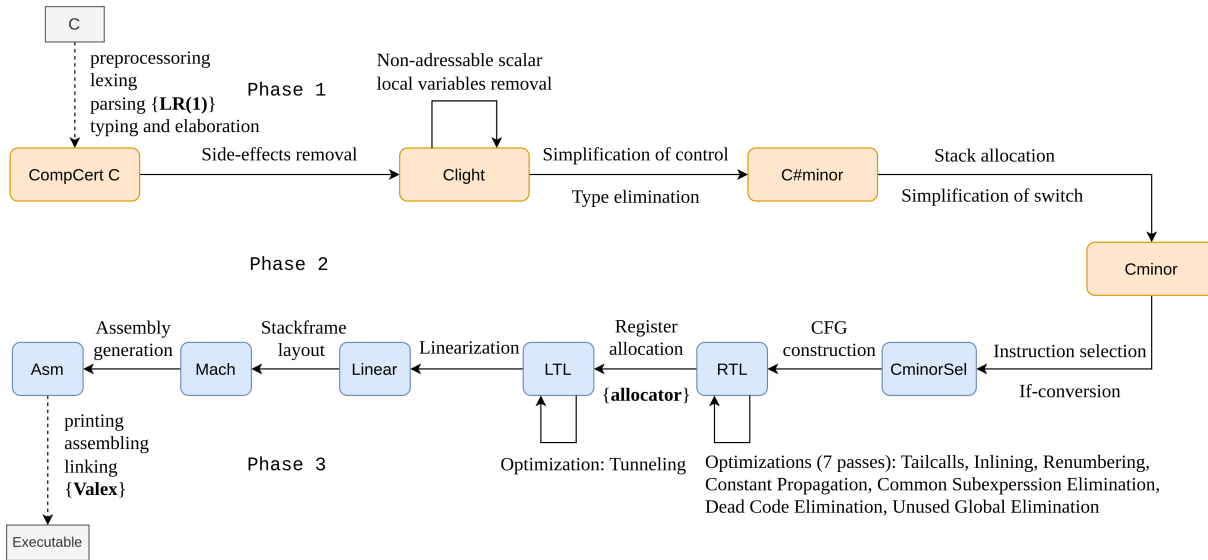


Figure 3.4 – CompCert Architecture

First, CompCert converts a C program into a precise, unambiguous CompCert C AST. The phase comprises preprocessing, lexing, parsing, type-checking, etc. Only the LR(1) parser is formally verified [JPL12].

The second phase is the most significant part of CompCert: a fully verified compilation process that transforms CompCert C AST into assembly ASTs. The front-end¹ translates a CompCert C AST into a Cminor AST, which is the lowest-level language still processor independent in the CompCert compilation chain. The back-end performs code transformation into a standard compiler IR, *i.e.*, register transfer language (RTL), performs optimizing passes, allocates registers by an external allocator written in OCaml, and finally obtains assembly code with various target architectures.

The last part consists of assembling and linking, executed by an external assembler and linker. While CompCert doesn't provide formal guarantees for this phase, the Valex tool from AbsInt uses translation validation to recheck the ELF executable files generated by the linker against the CompCert back-end assembly ASTs.

3.2.2 CompCert Programs

The syntax of programs in the all CompCert languages shares a common structure, shown in Figure 3.5.

1. The original paper of CompCert front-end[BDL06] starts from Clight, the version presented here simplifies our presentation.

Programs :

$$P ::= \{ \text{global_defs} = id_0 = Gd_0; \dots; id_n = Gd_n; \text{global definitions} \\ \text{pub_names} = id_0; id_1; \dots; id_n; \text{public names} \\ \text{main} = id \} \text{entry point}$$

Global definitions :

$$Gd ::= Gv \mid Fd$$

Definitions of global variables :

$$Gv ::= \{ \text{global_info} = v; \text{(language-dependent)} \\ \text{global_init} = data; \dots \} \text{initialization data, ...}$$

Function definitions :

$$Fd ::= \text{internal}(F) \mid \text{external}(Fe)$$

Definitions of internal functions :

$$F ::= \{ \text{sig} = sig; \text{body} = \dots; \dots \} \text{(language-dependent)}$$

Definitions of external functions :

$$Fe ::= \{ \text{name} = str; \text{sig} = sig \}$$

Function signatures :

$$sig ::= \{ \text{args} = \vec{ty}; \text{list of arguments type} \\ \text{res}(ty \mid \text{void}); \text{return type} \\ \text{cc} \} \text{calling convention}$$

Types :

$$ty ::= \text{int} \text{ integers and pointers} \\ \mid \text{float} \text{ floating-point numbers}$$

Figure 3.5 – Syntax of CompCert Programs

A program consists of

- `global_defs`: a list of global definitions, either global variables `Gv` with their initialization data or functions `Fd`;
- `pub_names`: a collection of public names that are visible outside of the program;
- `main`: the name of the main function that constitutes the program entry point.

CompCert supports two kinds of function definitions. Internal functions F are defined within the given language, and include at least i) a `signature` specifying the number and types of arguments and results, along with the additional information on which calling convention to use; ii) a `body` defining the computation task, *e.g.*, statements in C and instructions in assembly.

External functions Fe are defined outside the program, such as systems calls or compiler built-in functions, and are declared with an external name with string type (`str`) and a signature. The observable behavior of the program is defined in terms of a trace of invocations of external functions.

3.2.3 CompCert Memory Model

The memory model [LB08; Ler+12] and the representation of values are shared across all intermediate languages in CompCert. The set of values `val` is defined as follows:

$$\text{val} \ni v ::= \text{Vint}(i) \mid \text{Vlong}(i) \mid \text{Vptr}(b, o) \mid \text{Vundef} \mid \dots$$

A value $v \in \text{val}$ can be a 32-bit integer `Vint(i)`; a 64-bit integer `Vlong(i)`, a pointer `Vptr(b, o)` composed of a block identifier b and an offset o , or the undefined value `Vundef`. The undefined value `Vundef` represents an unspecified value and is not, strictly speaking, an undefined behavior. Yet, as most of the C operators are strict in `Vundef`, and because branching over `Vundef` or de-referencing `Vundef` are undefined behaviors. CompCert values also include floating-point numbers; they play no role in our development.

CompCert's memory consists of a collection of separate arrays, each with a fixed size determined at allocation time and identified by an uninterpreted block $b \in \text{block}$. The memory provides an API for basic operations:

- `alloc(M, lo, hi) = (b, M')`: Allocate a fresh block with bounds $[lo, hi]$, initially containing undefined cells. Return its identifies b and the updated memory M' .

-
- `load(k, M, b, ofs) = [v]`: Read the value v of memory chunk k from memory M , at block b , starting at index ofs .
 - `store(k, M, b, ofs, v) = [M']`: Store value v in memory chunk k of block b at offset ofs . Return the updated memory M' .
 - `free(M, b, lo, hi) = [M']`: Free the block b within the given bounds. Returns the updated memory M' where the interval $[lo, hi)$ of block b has been invalidated: not allowed to read or write anymore.

The memory chunk k involved in `load` and `store` specifies the number of bytes to be written or read and how to interpret bytes as a value $v \in \text{val}$. For instance, `Mint32` specifies a 32-bit value. The `alloc` operation never fails because CompCert assumes an infinite memory. `load` and `store` may fail when given an invalid block b , an out-of-bounds offset ofs , or invalid permissions, `free` may also fail because it requires freeable permission on the given range. Therefore, those operations return option types, with $[v]$ (*i.e.*, *Some v*), denoting success with result v , and \emptyset (*i.e.*, *None*) denoting failure.

3.2.4 CompCert Simulation Framework

CompCert is organized into passes that utilize several intermediate languages, as illustrated in [Figure 3.4](#). Each intermediate language is equipped with a formal semantics and each pass is rigorously proved to preserve the observational behavior of programs. For each pass, CompCert employs one of two verification approaches:

- Direct proof of correctness.
- Validation a posteriori (*i.e.*, translation validation) with proof of the validator’s correctness.

In this section, we primarily focus on the first approach, which is adopted by most CompCert passes. The translation validation technique is specifically applied in the transformation from RTL to LTL due to the ease of proving the validator’s correctness. For detailed information on the proof, we refer the interested reader to [\[RL10\]](#).

Transition Semantics. The operational semantics for all CompCert languages are defined as label transition systems. The transition relation, denoted as $G \vdash S \xrightarrow{t} S'$, represents a single execution step from state S to state S' within the global environment G . Each step is associated with a trace t of observable external I/O events (*e.g.*, a system

call) or an internal event ϵ . CompCert also defines transitive closures $\text{star} \rightarrow^*$ (zero, one or many steps) and $\text{plus} \rightarrow^+$ (one or many steps) from the one step \rightarrow . Additionally, each CompCert language defines two special predicates `initial_state` and `final_state` to represent when the execution of a program in the language starts and stops.

- `initial_state(P, S)`: Signifies that state S is an initial state for program P , typically corresponding to the invocation of the main function of P .
- `final_state(S, n)`: Indicates that S is a final state with an exit code n , implying that the program is returning from the initial invocation of its main function, producing value `Vint(n)`.

Generic Simulation Diagrams. CompCert uses a *simulation* approach to prove that each pass preserves the semantics between source and target languages. Two types of simulations are used in CompCert: forward simulation and backward simulation. Intuitively, a forward simulation asserts that every step in the source language corresponds to a set of steps in the target language, while a backward simulation states that each step in the target language corresponds to a set of steps in the source language.

Consider two languages, L_1 and L_2 , defined by their transition semantics. Let P_1 be a source program in L_1 , and P_2 be the translated target program in L_2 . We construct a relation $S \sim T$ between states of L_1 and states of L_2 and demonstrate that it constitutes either a forward or a backward simulation. To begin, we ensure that initial states and final states are related by \sim as follows:

- *Initial states*: if `initial_state(P1, S)` and `initial_state(P2, T)`, then $S \sim T$
- *Final states*:
 - (*forward*): if $S \sim T$ and `final_state(S, n)`, then `final_state(T, n)`.
 - (*backward*): if $S \sim T$ and `final_state(T, n)`, then `final_state(S, n)`.

Furthermore, assuming $S \sim T$, we relate transitions starting from S in L_1 with transitions starting from T in L_2 .

The standard technique is to prove that \sim is a backward simulation, as shown in [Figure 3.6](#). A backward simulation assumes that S from the source program P_1 and T from the target program P_2 preserve the simulation relation \sim , if the target P_2 performs one transition and results in the updated state T' , then there exists a new state S' obtained by corresponding many transitions (`star` or `plus`) of the source P_1 such that either the updated states S' and T' still preserve the same relation, or the measure $|T'|$ is less than the measure

$|T|$ in a well-founded ordering² and S' and T' preserve \sim . The backward simulation also requires the observable events align between the source transition and the corresponding target transitions. G_1 and G_2 represent the global environments corresponding to P_1 and P_2 , respectively. The measure strictly decreases to rule out the infinite stuttering case.

Theorem 1 (Backward Simulation).

$$\forall S T T', S \sim T \wedge G_2 \vdash T \xrightarrow{t} T' \Rightarrow \\ \exists S', (G_1 \vdash S \xrightarrow{t^+} S' \wedge S' \sim T') \vee (|T'| < |T| \wedge G_1 \vdash S \xrightarrow{t^*} S' \wedge S' \sim T')$$

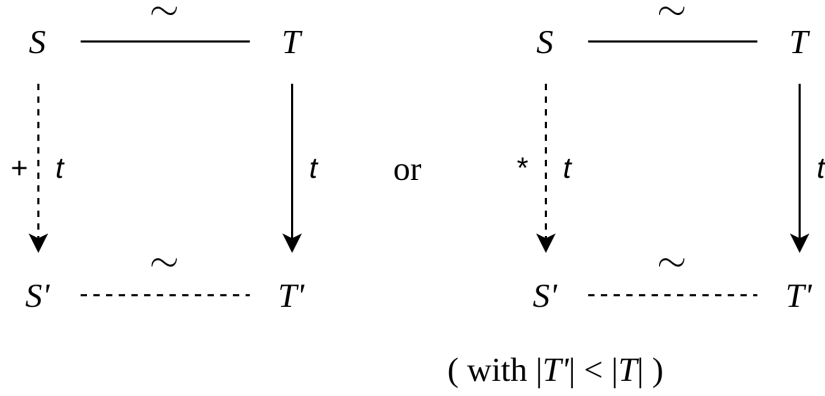


Figure 3.6 – CompCert Backward Simulation

An alternative to backward simulation is forward simulation, where every step in the source language is matched with a number of steps in the target language.

Theorem 2 (Forward Simulation).

$$\forall S T S', S \sim T \wedge G_1 \vdash S \xrightarrow{t} S' \Rightarrow \\ \exists T', (G_2 \vdash T \xrightarrow{t^+} T' \wedge S' \sim T') \vee (|S'| < |S| \wedge G_2 \vdash T \xrightarrow{t^*} T' \wedge S' \sim T')$$

CompCert also proves a crucial theorem that a forward simulation implies a backward simulation when the source language L_1 is *receptive* and the target language L_2 is *determinate*.

2. there are no infinite decreasing chains

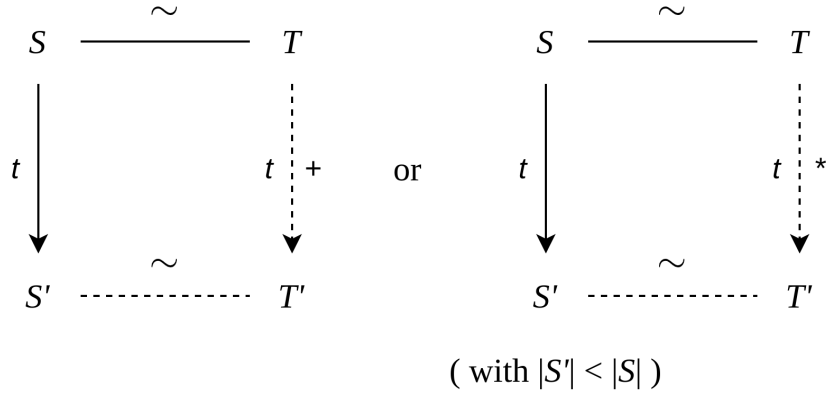


Figure 3.7 – CompCert Forward Simulation

Theorem 3 (Forward to Backward).

$$\forall L_1 L_2, \text{forward_simulation } L_1 L_2 \rightarrow \text{receptive } L_1 \rightarrow \text{determinate } L_2 \rightarrow \\ \text{backward_simulation } L_1 L_2$$

In this theorem, a language is considered *receptive* when progress is independent of the external environment, which holds for the languages of CompCert. A language is deemed *determinate* if a transition starting from a state S achieves two states S_1 and S_2 with the same trace t , then S_1 and S_2 are identical.

Program Behaviors. CompCert expresses the observable behaviors of programs in terms of traces of input-output events, where t represents finite traces and τ signifies an infinite trace:

$$bh ::= \text{Terminates}(t, n) \mid \text{Diverges}(t) \mid \text{Reacts}(\tau) \mid \text{GoesWrong}(t)$$

The behaviour $\text{Terminates}(t, n)$ corresponds to an execution that terminates normally, producing a return value n , after emitting a trace of events t . The behaviour $\text{Diverges}(t)$ represents an execution that emits a finite trace t but then runs forever without any observable events. $\text{Reacts}(\tau)$ is the behaviour of an execution that emits an infinite trace τ . The last behaviour GoesWrong corresponds to a program that becomes stuck (undefined behaviours) after having emitted the finite trace t .

The CompCert Theorem. CompCert starts with a *backward simulation* in the first pass (between CompCert C and Clight³) to deal with nondeterminism. Then, all subsequent passes from Clight to assembly are proved correct using *forward simulations* since backward simulations are generally more challenging to establish. CompCert results in a *backward simulation* lemma between Clight and assembly by following these steps: i) Composing the aforementioned forward simulations to form a *forward simulation*; ii) Utilizing [Theorem 3](#) to translate the *forward simulation* into the result.

Finally, the composition of the simulation lemmas across all CompCert passes forms the CompCert semantic preservation theorem.

Theorem 4 (CompCert’s Semantic Preservation). *Suppose that tp is the result of the successful compilation of the program p . If bh is a behaviour of tp then there exists a behaviour bh' such that bh' is a behaviour of p and bh improves on the behaviour bh' .*

$$\begin{aligned} \forall p \ tp \ bh, \text{compcert } p = [tp] \rightarrow bh \in \text{Asm.semantics}(tp) \rightarrow \\ \exists bh', bh' \in \text{Csem.semantics}(p) \wedge bh' \subseteq bh \end{aligned}$$

Where $bh' \subseteq bh$ if either bh' is equal to bh , or bh' is a undefined behaviour that is replaced by a defined behaviour in bh .

3.2.5 CompCert Ecosystem

As a cornerstone of formally verified compilers, CompCert inspires many formal verification projects, *e.g.*, CertiCoq, the verified operating system CertikOS [Gu+16], and C programming verification tool VST, etc. This section introduces some mature tools and projects related to CompCert, which are used or related to our work, as depicted in [Figure 3.8](#).

CompCert Tools. CompCert provides a range of essential tools for tracing and capturing immediate representation at various compilation passes. For instance, the CompCert [printCsyntax](#) module is used as a Cpretty-printer for CompCert C AST. Additionally, CompCert implements a mature tool [CLIGHTGEN](#) to extract formal representations of CompCert C or Clight syntax from informal C programs. Those unverified tools are writ-

3. In fact, CompCert introduces another IR named Cstrategy which is the same language with CompCert C but selects an evaluation strategy for expressions to guarantee determinism

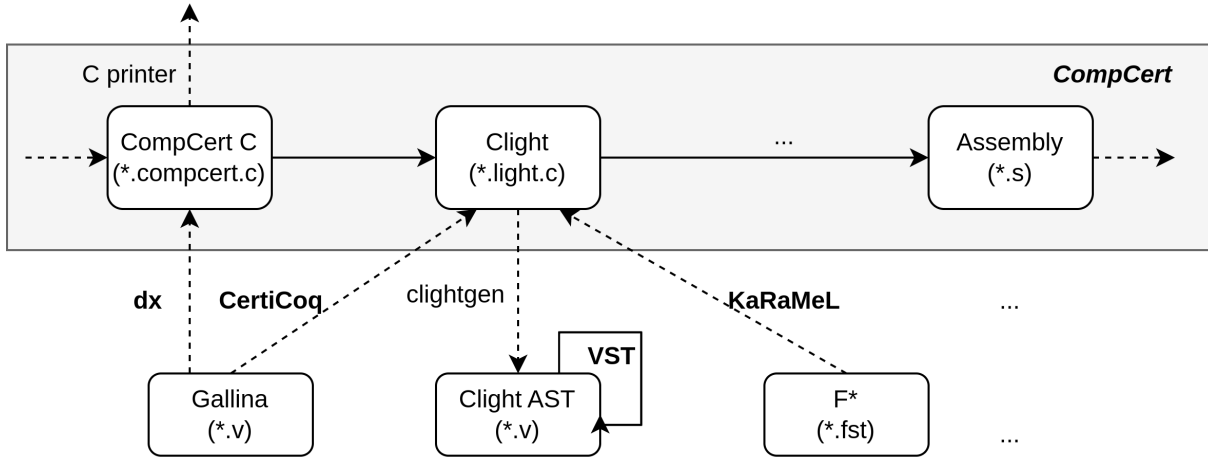


Figure 3.8 – CompCert tools and related projects

ten in OCaml, and used in many projects as trusted components, *e.g.*, the Verified Software Toolchain (VST) uses CLIGHTGEN to verify C programs at the Clight level.

CompCert Related Projects. As a verified C Compiler, CompCert serves as the backend for numerous verified and unverified compilers. For example,

- *KaRaMeL*: compiling a low-level subset of F* into the Clight language. The compilation is implemented in OCaml, and only has a theoretical proof in the paper [Pro+17].
- *CertiCoq*: translating Gallina, the function language of the Coq proof assistant, into Clight. As of the time of writing (until 2023-07-20), the verification process is ongoing.
- *∂x*: deriving CompCert C code from monadic Gallina programs. In our work, we use this unverified tool to obtain executable C code. We will provide further details about *∂x* in the next section.

3.3 ∂x Code Generator

The ∂x Tool. ∂x emerged from the toolchain used to design and verify the Pip proto-kernel [Jom+18b]. Since Pip’s source code is written in Gallina in a syntax close to that of C, ∂x naturally came as a need to translate Gallina code to C code. ∂x extracts C code from a Gallina source program in the form of a CompCert C AST. The goal of ∂x is to

provide C programmers with readily reviewable code and thus avoid misunderstanding between those working on C/assembly modules (that access hardware) and those working on Coq modules (the code and proofs). To achieve this, ∂x handles a C-like subset of Gallina. The functions that are to be converted to C rely on a monad to represent the side effects of the computation, such as modifications to the CPU state. Yet ∂x does not mandate a particular monad for code extraction.

∂x 's Workflow. ∂x proceeds in three steps, as shown in Figure 3.9 (colored in pink). The first step (*Coq_to_IR*) generates an IR for the subset of Gallina ∂x can handle, given a list of Gallina functions, or whole modules. Since Coq has no built-in reflection mechanism, this step is written in Elpi [Dun+15], using the Coq-Elpi plugin [Tas21]. The second step (*IR_to_C*) is to translate this IR into a CompCert C AST. This step can also process external functions (appearing as `extern` in the extracted C code) to support separate compilation with CompCert. In order to obtain an actual C file, the last step of ∂x (*DumpAsC.ml*) provides a small OCaml function that binds the extracted C AST to CompCert's C pretty-printer.

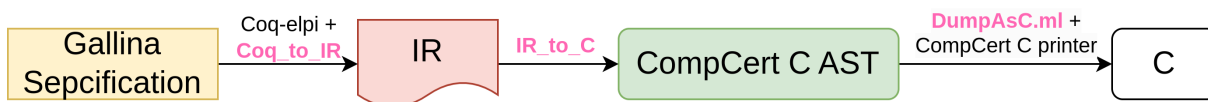


Figure 3.9 – ∂x workflow

Even though the ∂x language is a small subset of Gallina, it inherits much expressivity from the use of Coq types to manipulate values. For example, we can use bounded integers (*i.e.*, the dependent pair of an integer with the proof that it is within some given range), for instance CompCert's `int`, that can be faithfully and efficiently represented as a single `int` in C. To this end, ∂x expects a configuration mapping Coq types to C.

∂x Memory Management. A major design choice in the C-like subset of Gallina used by ∂x is memory management: its generated code executes without garbage collection. This affects the Coq types that can actually be used in ∂x : recursive inductive types, such as, lists cannot automatically be converted⁴. However, this Gallina subset is particularly relevant to programs in which one wants to precisely control memory management

4. ∂x requires users to leverage lists in a constrained way: *e.g.*, the lists are pre-allocated and their sizes are fixed so that there is no allocation or de-allocation appearing in the program body.

and decide how to represent data structures in memory. This is typically the case of an operating system or, in our case, the rBPF virtual machine.

∂x **Example.** We illustrate the *∂x* workflow using the *sumarray* example. The specification of *sumarray* is as follows:

“calculating the sum of elements in an integer array”

Firstly, *∂x* requires a monadic Gallina specification with arbitrary monads. In this case, we utilize an option state monad M with two standard operators *returnM* and *bindM*. The state is defined as a list of CompCert integers, represented as *list int* in Gallina. The option type is used to capture any array out-of-bound errors. For improved readability, we adopt the standard notation ‘*do x ← a ; b*’ to represent ‘*bindM a (fun x => b)*’.

```

Definition state := list int.

Definition M (A: Type) := state -> option (A * state).
Definition returnM {A: Type} (a: A) : M A := fun s => Some (a, s).
Definition bindM {A B: Type} (x: M A) (f: A -> M B) : M B := fun s =>
  match x s with
  | None => None
  | Some (x', s') => (f x') s'
end.

Declare Scope monad_scope.
Notation "'do' x <- a ; b" := (bindM a (fun x => b))
  (at level 200, x name, a at level 100, b at level 200) : monad_scope.

```

Next, the memory management for *sumarray* in Gallina adopts an abstract method: modelling arrays into Gallina Lists with a fixed size. Consequently, all memory operations extend standard Gallina list functions with a monadic effect, *e.g.*, a C array-access operation *get* is defined as the function ‘*List.nth_error : ∀ A : Type, list A → nat → option A*’.

As a result, we define the monadic Gallina specification of *sumarray* as follows: it takes the length of the array, *i.e.*, *len* with the natural number type *nat*, and invokes a tail-recursive *sumarray_aux* function to traverse the array, starting from the last element. The *sum* argument initializes to 0. The *sumarray_aux* function performs a case analysis

on the construction of the natural number *index*. If *index* reaches to Peano number zero (denoted `0` or `0` in Coq), the function returns the result *sum*. Otherwise *index* has the form of *n* with a successor operation (denoted `S` in Coq), `sumarray_aux` accesses the *n*-th element of the list, counts its value *v* into the result *sum*, and does the next computation with arguments *n* and the new result.

```

Open Scope monad_scope.
Definition get (index: nat): M int := fun st =>
  match List.nth_error st index with
  | Some v => Some (v, st)
  | None => None
  end.

Fixpoint sumarray_aux (index: nat) (sum: int): M int :=
  match index with
  | 0 => returnM sum
  | S n => do v <- get n;
           sumarray_aux n (Int.add sum v)
  end.

Definition sumarray (len: nat): M int := sumarray_aux len 0%nat
Close Scope monad_scope.

```

Consequently, it's necessary to configure ∂x for mapping Coq types into CompCert C types and mapping Coq operators into CompCert C expressions. We designate `C_U32` as the 32-bit integer type in the CompCert C syntax which represents *unsigned int* of C. ∂x maps Coq *nat* type to a C finite fix-sized type since the hardware cannot represent infinite natural numbers. In this case, we select *e.g.*, *unsigned int* (see `natCompilableType`) as we carefully assume the size of our array is less than 2^{32} . We also declare the CompCert integer *int* in Coq should be translated into `C_U32` in the CompCert C syntax (see `intCompilableType`). For the CompCert addition function `Int.add`, we define the configuration `Const_Int_add` to map it into the corresponding expression, denoted as `C_add`, in the CompCert C syntax. For simplification, we omit its implementation details.

```

Definition natCompilableType := (**r mapping `nat` to `unsigned int` *)
  MkCompilableType nat C_U32.
Definition intCompilableType := (**r mapping `CompCert int` to `unsigned int` *)
  MkCompilableType int C_U32.

```

```

Definition C_add (x y: Csyntax.expr): Csyntax.expr :=
  Csyntax.Ebinop Cop.Oadd x y C_U32.

```

```

Definition Const_Int_add := ... (**r Int.add x y -> C_add c_x c_y *)

```

Finally, ∂x generates the corresponding C code from the monadic *sumarray* Gallina function based on the monad and configuration. ∂x follows a syntax-directed code generation approach, translating the Gallina functions `sumarray_aux` and `sumarray` into corresponding C code with the same function name, nearly identical argument names, and a similar code structure. The generated C functions include an additional argument called *st*, representing the global monadic state. Coq types *nat* and *int* are translated into C's *unsigned int*, and the Coq *match-pattern* on *nat* is transformed into an if structure (line 6 - line 12). ∂x can only translate the two standard monad operators, where `returnM` is converted into a `return` statement in C, and `bindM` is translated into a C assignment statement.

```

1 //arraysum.c
2 unsigned int sumarray_aux(unsigned int* st, unsigned int index, unsigned int sum)
3 {
4     unsigned int n;
5     unsigned int v;
6     if (index == 0U) {
7         return sum;
8     } else {
9         n = index - 1U;
10        v = get(st, n);
11        return sumarray_aux(st, n, sum + v);
12    }
13 }
14
15 unsigned int sumarray(unsigned int* st, unsigned int len)
16 {
17     return sumarray_aux len 0U;
18 }

```

∂x translates all non-standard functions as C external functions, *e.g.*, *get*. These external functions are replaced with manual C implementations declared in the header file.

```
1 //arraysum.h
2 static inline unsigned int get(unsigned int* st, unsigned int index) {
3     return st[index];
4 }
```

3.4 Conclusion

In this chapter, [Section 3.1](#) introduces the BPF family, including classical BPF, Linux extended BPF, and our target RIOT-OS rBPF. Subsequently, [Section 3.2](#) provides a brief overview of the CompCert architecture, CompCert memory model, and the CompCert simulation framework. We also introduce the CompCert tools used in this document and projects related to our work in [Section 3.2.5](#). Lastly, [Section 3.3](#) briefly presents the ∂x workflow and employs an example to illustrate how ∂x operates.

AN END-TO-END VERIFICATION APPROACH IN COQ

This chapter firstly discusses the path selection of formal techniques for verifying rBPF, then presents an overview of our methodology for deriving a verified C implementation from a Gallina specification, and finally links the proposed methodology to its applications.

4.1 Discussion: Which Way Do We Select

At the beginning, our verification object is an unverified RIOT-OS rBPF VM written in C. Our task is to formally prove that it satisfies some critical properties.

As introduced in [Section 1.1](#), one possible verification path adds annotations (*e.g.*, pre/post conditions) to the original C code for describing program invariants and expected properties, for instance, F*, Frama-C or RefinedC. Then, those formal annotations/asserts can be automatically checked by some backend SMT solvers. Although this path benefits from a high degree of proof automation, it has a large TCB of SMT solvers and also has less expressiveness compared to proof assistants like Coq. We used F* to verify another RIOT-OS component bootloader (approx. 100 loc), named *riotboot* [YT21], and the expected properties are functional correctness and memory safety. Our previous experiences show this path requires more potential proof efforts when SMT solvers cannot solve the input asserts; verification engineers need to spend much time figuring out the reason because solvers works as black boxes.

Another possible option is to use proof assistants to verify the original C code, for example the VST tool. VST uses CompCert CLIGHTGEN tool to extract Clight AST; then users declare the expected properties as pre/post conditions and prove the final theorems with VST-specific tactics in Coq. The VST solution has a small TCB because it is built on the Coq checker. The cost is that it requires more proof efforts as Coq has less automation compared to SMT solvers, and low-level details (*e.g.*, memory operations

and optimizations) also increase the difficulty and complexity of proofs.

In this thesis, we adopt an end-to-end verification approach: we formalize a high-level specification of the rBPF VM, making the proof of expected properties significantly easier due to abstraction. Subsequently, We use the verified compilation to propagate these properties down to a low-level C implementation. To enhance efficiency, we introduce step-wise refinements as intermediate models. As discussed in [Chapter 2](#), existing end-to-end verification do not simultaneously meet our requirements, *i.e.*, verified compilation, efficient C code, and small TCB. Therefore, we propose an end-to-end verification workflow.

4.2 A Workflow for End-to-End Verification in Coq

Our approach provides an end-to-end correctness proof, within the Coq proof assistant, that reduces the hurdle of reasoning directly over the C code. In the subsequent chapters, the methodology will be instantiated to derive the C implementation of a fault-isolating rBPF virtual machine and its verifier.

As shown in [Figure 4.1](#), the original object implementation (rBPF C code) or specification (JIT specification) is first formalized by a proof model in Gallina, and the verification of expected properties (*e.g.*, safety) is performed within the Coq proof assistant. This formal specification is then refined into an optimized (and equivalent) synthesis model ready for C-code extraction.

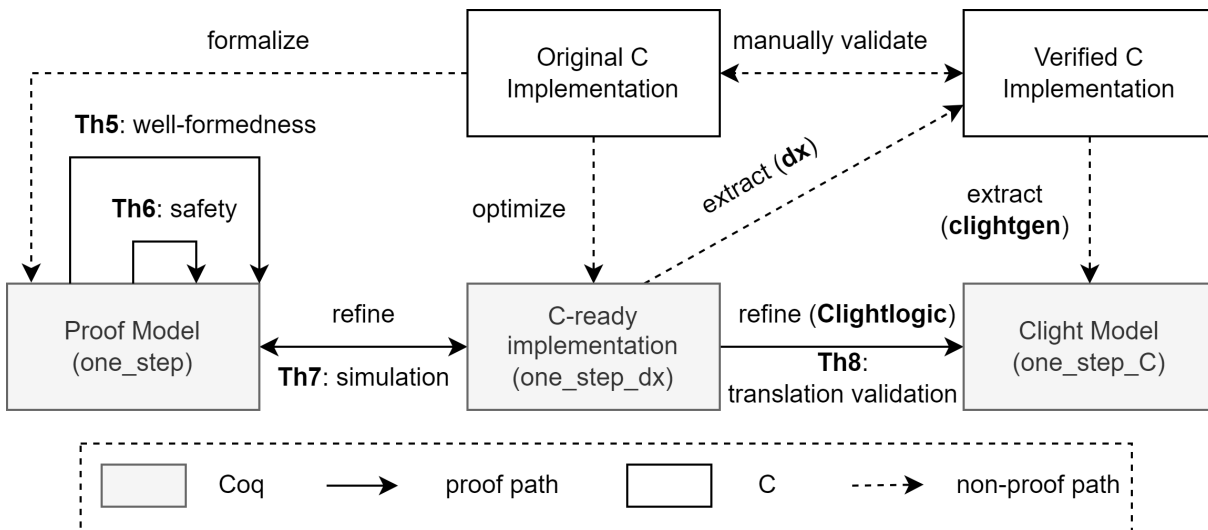


Figure 4.1 – End-to-end verification and synthesis workflow

Our refinement and optimization principle aims to derive a C-ready implementation

in Gallina that is **as close as possible to the expected target C code**. This principle serves several purposes:

1. It allows us to prove optimizations correct.
2. It improves the performance of the extracted code and.
3. It facilitates the review and validation of extracted code with the system designers.

From the C-ready Gallina implementation, we leverage ∂x to automatically generate C code and verify it: i) The generated C code is first parsed as a CompCert Clight model using the CLIGHTGEN tool. ii) We then prove that the generated C code refines the source Gallina model in Coq. Because ∂x generates C code in a syntax-directed manner, a minimal *Clightlogic* is designed to facilitate the refinement proof. The rest of this section explains these different steps in details.

4.2.1 Proof-Oriented Specification

Our approach starts with a proof-oriented specification in Gallina, which consists of three important aspects:

- *monad*: The specification is structured as an executable abstract machine in monadic form to capture side effects *e.g.*, memory store operations.
- *invariants*: The machine preserves a collection of invariants, *i.e.*, *well-formedness* defined below, for the purpose of safety proof.
- *safety*: The safety property of the machine is defined, ensuring that a well-formed machine never leads to any error.

Firstly, our specification uses the standard option-state monad M .

Option-state Monad. M takes an input state and may return a result of type a along with a new state. The option type is used to capture undefined behaviors, represented as *None*. M provides two primary operations: $returnM$ for retrieving the state and $bindM$

for modifying it with a new state.

$$\begin{aligned}
M \text{ a state} &:= \text{state} \rightarrow \mathbf{option}(a \times \text{state}) \\
\text{return}M &: a \rightarrow M \text{ a state} := \lambda a. \lambda st. \llbracket (a, st) \rrbracket \\
\text{bind}M &: M \text{ a state} \rightarrow (a \rightarrow M \text{ b state}) \rightarrow M \text{ b state} := \\
&\lambda A. \lambda f. \lambda st. \\
&\mathbf{match} \ A \ st \ \mathbf{with} \\
&| \emptyset \Rightarrow \emptyset \\
&| \llbracket (x, st') \rrbracket \Rightarrow (f \ x) \ st'
\end{aligned}$$

As explained in [Section 3.2.3](#), $\llbracket v \rrbracket$ (*i.e.*, *Some v*) denotes success with result v , and \emptyset (*i.e.*, *None*) denotes failure. In the remainder, we write ‘ $M \ a$ ’ for ‘ $M \ \text{a state}$ ’ when we know what the state represents.

The monad threads the state along computations to model in-place update. The safety property of the machine is implemented as an inline monitor: any violation leads to an unrecoverable error, *i.e.*, the unique error represented by \emptyset . One step of the machine has the following signature:

$$\text{one_step} : M \ \text{unit} \ \text{state}$$

where *unit*, a singleton datatype in Coq, denotes *one_step* returns an empty result and a new state. The *one_step* function implements a defensive semantics, checking the absence of error, dynamically. For our rBPF interpreter (see [Section 5.1](#)), the absence of error ensures that the rBPF code only performs valid instructions. In particular, all memory accesses are restricted to a sandbox specified as a list of memory regions. Function *one_step* is part of the TCB and, therefore, a mis-specification could result, after refinement, in an invalid computation. The purpose of the error state is to specify state transitions that would escape the scope of the safety property and, therefore, shall never be reachable from a well-formed state $st \in wf \subseteq \mathcal{P}(\text{state})$.

Well-formedness. We require well-formedness to be an inductive property of the *one_step* function.

Theorem 5 (Well-formedness). *The one_step function preserves well-formedness.*

$$\forall st, st', r. st \in wf \wedge \text{one_step} \ st = \llbracket (r, st') \rrbracket \Rightarrow st' \in wf$$

Intuitively, well-formedness represents a collection of assumptions about the internal

invariants and external environments of the machine, for example, the invariants of loop statements, and the hardware information (endianness, 32- or 64-bit architecture, etc).

Safety. We also require that well-formedness is a sufficient condition to prevent the absence of error, ensuring the safety of computations.

Theorem 6 (Safety). *The `one_step` function is safe, i.e., a well-formed state never leads to an error.*

$$\forall st. st \in wf \Rightarrow one_step\ st \neq \emptyset$$

4.2.2 C-ready implementation

Our methodology involves refining the `one_step` function into an optimized `one_step ∂x` that complies with the requirements of ∂x . The refinement consists of two steps:

1. The first step of refinement involves implementing optimization strategies derived from the original implementation or related specifications. As ∂x performs syntax-directed code generation, the efficiency of the extracted code crucially depends on `one_step ∂x` .
2. The second step is to incorporate ∂x configurations into the optimized Gallina model. This allows ∂x to identify input Gallina functions and extract them into the corresponding C functions with expected type signatures and code structure.

To ensure the absence of errors, we establish a simulation relation between the `one_step` and `one_step ∂x` functions. A direct consequence of the simulation theorem is that `one_step ∂x` never raises an error.

Theorem 7 (Simulation). *Given simulation relations $R_s \subseteq state \times state'$ and $R_r \subseteq r \times r'$, the function `one_step ∂x` simulates the function `one_step`.*

$$\forall s_1, s'_1, s_2, r. (s_1, s_2) \in R_s \wedge one_step\ s_1 = [(r, s'_1)] \Rightarrow \exists s'_2, r'. \bigwedge \begin{cases} one_step_{\partial x}\ s_2 = [(r', s'_2)] \\ (s'_1, s'_2) \in R_s \\ (r, r') \in R_r \end{cases}$$

4.2.3 Translation Validation of C code

The next stage consists in refining the `one_step ∂x` function into a Clight program using ∂x to get a C program, and the CLIGHTGEN tool to obtain a Clight `one_stepC`

program (see [Section 5.2](#)). As this pass is not trusted, we require the following translation validation theorem.

Theorem 8 (Translation Validation). *Given a simulation relation $Rs \subseteq \text{state}' \times \text{val} \times \text{mem}$ and a relation $Rr \subseteq \text{res} \times \text{val}$, the Clight code one_step_C refines the function $\text{one_step}_{\partial x}$:*

$$\begin{aligned} \forall r, s, s', v, k, m. (s, v, m) \in Rs \Rightarrow \text{one_step}_{\partial x} s = \lfloor (r, s') \rfloor \Rightarrow \\ \exists m', r'. \text{Callstate}(\text{one_step}_C, [v], k, m) \rightarrow^{*t} \text{ReturnState}(r', \text{call_cont}(k), m') \wedge \\ (s', v, m') \in Rs \wedge (r, r') \in Rr \end{aligned}$$

[Theorem 8](#) states that, if $\text{one_step}_{\partial x} s$ runs without error and returns a result (r, s') , then, the Clight function one_step_C successfully runs with argument v and, after a finite number of execution steps, returns a result r' and a memory m' that preserve the refinement relations. In our encoding, the unique argument v is a pointer to the memory allocated region refining the interpreter state and k represents the continuation of the computation. A corollary of [Theorem 8](#) is that the Clight code one_step_C is free of undefined behaviors. In particular, all memory accesses are valid. As the memory model does not allow to forge pointers, this yields a strong isolation property.

4.2.4 Summary

In summary, this section presents our methodology, which is an end-to-end verification approach from monadic Gallina functions to executable C implementations. The verification effort has several levels:

- The abstract Gallina model is proven to preserve well-formedness and safety properties.
- The optimized Gallina implementation is proven that it has the same program behaviors as the abstract one, which implies that it also preserves the absence of errors.
- The final extracted C code is also behavior-equivalent with the original Gallina model. This is achieved by proving a translation validation theorem between its Clight model and the input model of ∂x .

In the remainder of this document, for our rBPF virtual machine, we prove all the aforementioned properties within the Coq proof assistant.

4.3 Applications

This section gives a technical overview of CertrBPF and CertrBPF-JIT to establish a connection between our end-to-end verification methodology to its practical applications.

CertrBPF interpreter. Firstly, we apply our end-to-end verification approach to the interpreter of the rBPF VM.

- In [Section 5.1](#), we formalize the rBPF interpreter in monadic form and prove (corresponding to [Theorem 6](#)) that this monadic model never crashes with respect to invariants (corresponding to [Theorem 5](#)) about rBPF register, memory, verifier, etc.
- [Section 5.2](#) explains how to refine the proof model into an optimized one with additional ∂x configuration. The refined model is proven to be equivalent to the proof model (related to [Theorem 7](#)).
- In [Section 5.3](#), we introduce the *Clightlogic* and prove [Theorem 8](#).

CertrBPF verifier. Secondly, we apply the methodology to the verifier of rBPF VM. We implement a proof model for the verifier and prove its success with a boolean result (corresponding to [Theorem 6](#)). As the rBPF verifier only checks some basic rules, the verifier synthesis model is refined only with ∂x configuration. The related proofs and C code extraction are omitted for simplification, their Coq code can be found in the repository.

CertrBPF JIT. Lastly, we apply the end-to-end verification approach to the JIT compiler of rBPF VM. The current development status includes a proof model, a synthesis model, and the extracted C implementation, the complete proof will be completed in a timely manner.

CERTRBPF: A FULLY VERIFIED rBPF VIRTUAL MACHINE

Implementing a fault-isolating virtual machine for MCUs faces two significant challenges. The first is to embed the VM within the MCU’s micro-kernel while minimizing its code size and execution environment. The second is to minimize the verification gap between its proof model and the running code.

In this chapter, we address these challenges and present the first end-to-end verification and synthesis of a full-scale, real-world virtual machine for the BPF instruction set family: CertrBPF, an interpreter tailored to the hardware and resource constraints of MCU architectures running the IoT operating system RIOT-OS.

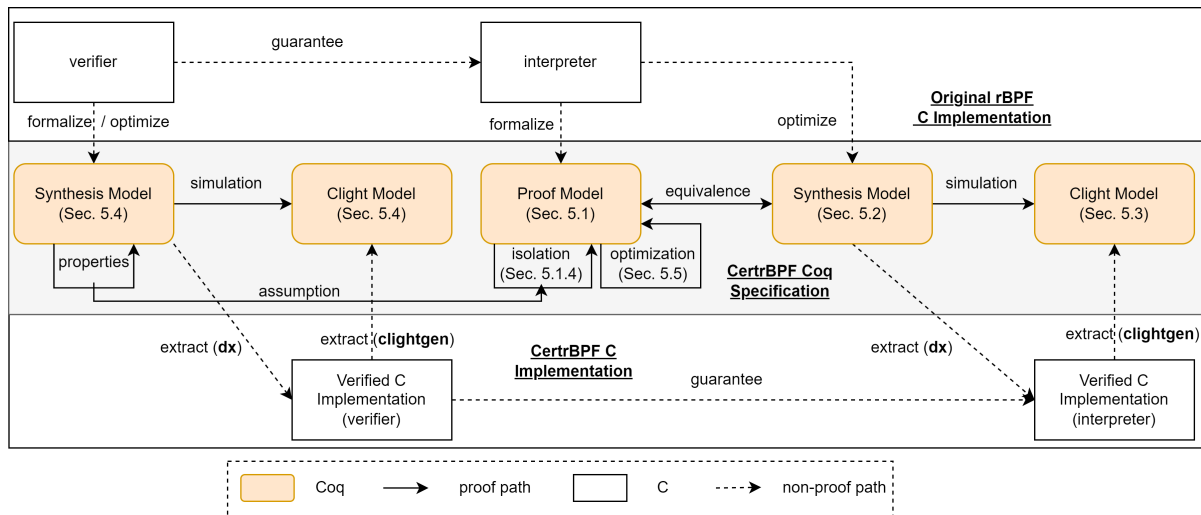


Figure 5.1 – Chapter Structure: CertrBPF

CertrBPF adopts the end-to-end verification workflow introduced in [Chapter 4](#). Our starting point is a model of the rBPF semantics written in Gallina ([Section 5.1](#)), as shown in [Figure 5.1](#). Using this proof model, we certify that all the memory accesses are valid and isolated to dedicated memory areas, thus ensuring isolation ([Section 5.1.4](#)). From

this proof model, we then derive a synthesis C-ready model that is optimized, safe, and behaviorally equivalent (Section 5.2). Additionally, we define a reusable ∂x library for mapping CompCert types and operations into some standard C types and operators. Finally, we extract an executable Clight implementation from the C-ready model and prove to perform the same state transitions (Section 5.3).

BPF families, *e.g.*, Linux eBPF and RIOT-OS rBPF, always require a static analyzer to guarantee that BPF binary programs satisfy critical properties *e.g.*, all jump instructions never cause *out-of-bound branches*. Following the same verification workflow, we formalize a CertrBPF verifier in Gallina, prove its correctness, and extract to an executable C implementation by reusing the ∂x - CompCert configuration (Section 5.4).

Last, we discuss a CertrBPF optimization (Section 5.5) and conclude (Section 5.6).

5.1 A Proof-Oriented Virtual Machine Model

Our proof model of rBPF includes explicit syntax, interpreter state, and monadic semantics functions, particularly those implementing dynamic security checks.

5.1.1 Syntax

The formal syntax of rBPF \mathfrak{P} is given in Figure 5.2, The rBPF instruction set features unary (*i.e.*, negation) and binary arithmetic operations, conditional or unconditional (*i.e.*, always jump *ja*) relative to an offset, operations to load values from memory to registers, operations to store values from registers to memory, function calls, and termination.

There are four kinds of operands available for the rBPF ISA:

1. A signed immediate number with a 32-bit width to represent a source value.
2. A signed 16-bit value to specify the jump offset, used only by branch instructions.
3. A 64-bit register operand ($\in \{R_0, \dots, R_9\}$)¹, usually for a destination operand.
4. A 64-bit register ($\in \{R_0, \dots, R_{10}\}$) or an immediate operand for a source operand.

rBPF also specifies arithmetic, logic, shift operators for binary arithmetic instructions, unsigned and signed conditional operators for branch instructions, and four memory chunks for memory instructions. For instance, *eq*, in symbol '=' , represents jump if (unsigned) equal, *set*, in symbol '&' , represents jump if signed equal, and *word* indicating 4 bytes operated by memory instructions.

1. *R10* is a read-only pointer to the bottom of a 512B stack.

Registers :	
$Regs ::= R_0 \mid R_1 \mid R_2 \mid R_3 \mid R_4$	General-purpose
$R_5 \mid R_6 \mid R_7 \mid R_8 \mid R_9$	General-purpose
R_{10}	Stack Frame Pointer
Operands :	
$imm \in Immediate$	Signed 32-bits immediate
$ofs \in Offset$	Signed 16-bits offset
$dst, reg \in Regs$	Register operand
$src \in Regs \cup Immediate$	Register or Immediate
Operators :	
$op ::= add \mid sub \mid mul \mid div \mid mod \mid mov$	Arithmetic operators
$and \mid or \mid xor$	Logic operators
$lsh \mid rsh \mid arsh$	Shift operators
$cmp ::= eq \mid neq \mid lt \mid gt \mid le \mid ge$	Unsigned conditional operators
$set \mid slt \mid sgt \mid sle \mid sge$	Signed conditional operators
Memory Chunk :	
$chk ::= byte \mid halfword \mid word \mid doublewords$	1/2/4/8 Bytes
Instruction :	
$ins ::= Neg \ dst$	Unary arithmetic
$Alu32 \ op \ dst \ src$	32-bit Binary arithmetic
$Alu64 \ op \ dst \ src$	64-bit Binary arithmetic
$Ja \ ofs$	Unconditional branch
$Jump \ cmp \ dst \ src \ ofs$	Conditional branch
$Load \ chk \ dst \ reg \ ofs$	Memory load
$Store \ chk \ dst \ src \ ofs$	Memory store
$Call \ imm$	Call
$Exit$	Return

Figure 5.2 – Core syntax of rBPF instruction set

5.1.2 Machine State

We define an rBPF semantic state as an 8-tuple $rbpf_state ::= (C, C_len, PC, R, f, M, MRs, mrs_num)$ 🍷 :

- C : A sequence of rBPF instructions in the form of 64-bit binaries.
- C_len : The fixed length of the input rBPF instructions.
- PC : The location of the currently executing instruction, *i.e.*, the Program Counter (PC) of rBPF.
- R : rBPF register map with 11 registers.
- f : An interpreter flag indicating the status of the rBPF VM.
- M : The CompCert memory.
- MRs : A specification of user memory regions declared by the input rBPF programs.
- mrs_num : The fixed number of available memory regions.

The specification of rBPF derives from Linux eBPF, whose ISA only declares 10 general purpose registers ($\{R_0, \dots, R_9\}$) and a read-only frame pointer register R_{10} . The original C implementation of the rBPF interpreter uses a pointer to the current instruction located in the input BPF binary list. Here we declare PC as a field of the machine state to record the index of the current instruction in the binary list.

The flag f characterizes the state of the rBPF interpreter, which can be: i) A normal state, written f_n . ii) A final state, written f_t . iii) Or an error state, written f_e . The error state f_e indicates that the defensive checks of the interpreter have detected an impending invalid behavior. For example, each 64-bit binary instruction leaves 4-bit for representing a destination register with the range $[0, 15]$, and a valid register must be in $[0, 10]$. Therefore a register-out-of-range error may occur if the destination field is '1110' which represents an illegal register R_{14} .

In contrast to Linux eBPF, rBPF requires users to explicitly declare all used memory regions by their BPF programs. Formally, a memory region 🍷 $mr = \langle start, size, p, ptr \rangle \in MRs$ associates a permission $p \in \{Readable, Writable\}$ with the address range $[start, start + size)$. As shown in [Figure 5.3](#), we establish the link between concrete physical addresses and the CompCert memory model using the pointer $ptr (= \mathbf{Vptr}(b, 0))$ where the block b is the abstract representation of the address $start$.

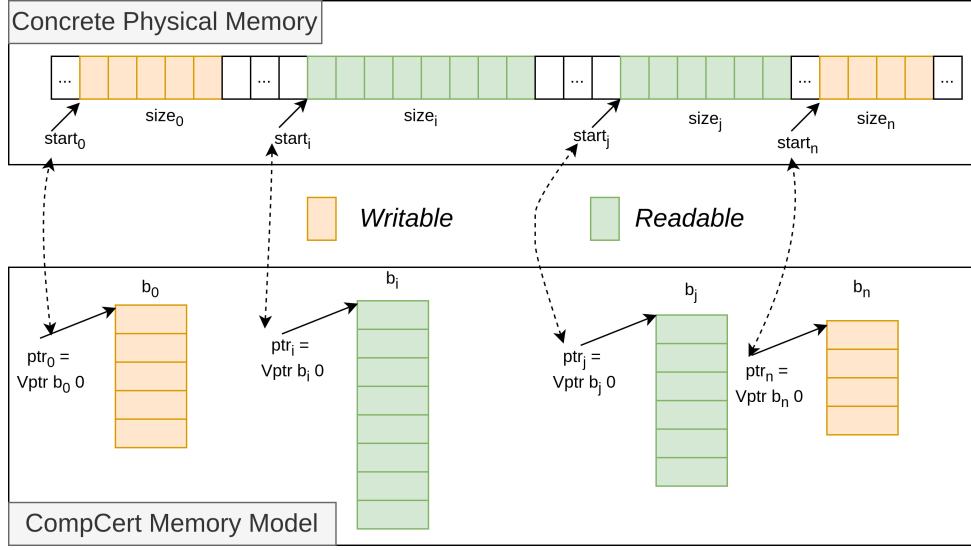


Figure 5.3 – Maps between (C) physical memory and (CompCert) memory model.

5.1.3 rBPF Interpreter

To formalize the behaviors of the rBPF interpreter, we first introduce some notations and function signatures, then describe the specifications of dynamic checks for the defensive purposes of our interpreter, and finally define the semantics of the interpreter using monadic functions.

We write $C[PC]$ for the instruction located at the program counter PC and $R[r] : \text{int64}$ to retrieve the 64-bit value of register r in the register map R . The function $\text{alu32} : \text{op} \rightarrow \text{val} \rightarrow \text{val} \rightarrow \text{option val}$ formalizes the formal semantics of rBPF 32-bit binary arithmetic operators. It takes an rBPF binary operator and two CompCert Values as input and returns an option CompCert value. The formal behaviours involves casting values from 64-bit to 32-bit, performing computation using CompCert’s operators on val types, and casting the result back from 32-bit to 64-bit. The alu32 function involves the following sub-functions:


- Val.longofintu : Type casting from 32-bit unsigned integer to 64-bit.
- val_intuoflongu : Type casting from 64-bit unsigned integer to 32-bit.
- $\text{Val.add}/\text{Val.divu}$: Standard CompCert Value operations on 32-bit integers. Val.divu returns \emptyset in case of *division-by-zero*.
- $\text{option_map} : (f : A \rightarrow B) \rightarrow (\text{option } A) \rightarrow (\text{option } B)$: Applies a function f to the **Some** case of the initial option type and return a new option type.

The function `alu64` has the same type signature as `alu32`, but its semantics are much simpler, involving straightforward use of CompCert’s 64-bit value operations.


Similarly, the function `cmp : cmp → val → val → bool` is a simplified version of CompCert’s `Val.cmplu_bool`, as rBPF only compares two 64-bit integers. For simplification, we omit the details of `cmp`.

$$\begin{aligned}
 \text{alu32}(\text{bop}, v_1, v_2) = & \\
 \left\{ \begin{array}{ll}
 \lfloor \text{Val.longofintu} (\text{Val.add} (\text{val_intuoflongu } v_1) (\text{val_intuoflongu } v_2)) \rfloor & \text{if } \text{bop} = \text{add} \\
 \text{option_map} (\text{fun } x \Rightarrow \text{Val.longofintu } x) & \\
 (\text{fVal.divu} (\text{val_intuoflongu } v_1) (\text{val_intuoflongu } v_2)) & \text{if } \text{bop} = \text{div} \\
 \dots & \dots
 \end{array} \right.
 \end{aligned}$$

$$\text{alu64}(\text{bop}, v_1, v_2) = \left\{ \begin{array}{ll}
 \lfloor \text{Val.addl } v_1 \ v_2 \rfloor & \text{if } \text{bop} = \text{add} \\
 \text{Val.divlu } v_1 \ v_2 & \text{if } \text{bop} = \text{div} \\
 \dots & \dots
 \end{array} \right.$$

Dynamic Checks. The function `check_alu`  dynamically checks the validity of an arithmetic instruction to avoid *div-by-zero* and *undefined-shift* errors. For division instructions, `check_alu` mandates the second argument to be non-zero. For arithmetic and logical shift instructions, the second argument must be below $n \in \{32, 64\}$, depending on whether the ALU instruction operates on 32 or 64-bit operands. This section discusses 64-bit ALU instructions but CertrBPF also includes the 32-bit ALU instructions.

$$\text{check_alu}(\text{op}, v) = \left\{ \begin{array}{ll}
 v \neq 0 & \text{if } \text{op} \in \{\text{div}, \text{mod}\} \\
 0 \leq v < n & \text{if } \text{op} \in \{\text{lsh}, \text{rsh}, \text{arsh}\} \\
 \text{true} & \text{otherwise}
 \end{array} \right.$$

The function `check_mem`  returns a valid pointer (`Vptr(b, ofs)`) if there exists a unique memory region $mr \in MRs$ such that

1. The permission $mr.perm$ is at least *Readable* for **Load** and *Writable* for **Store**, *i.e.*, $mr.perm \geq p$.
2. The offset ofs is aligned, *i.e.*, $ofs \% Z(\text{chk}) = 0$ ².
3. ofs is within bounds, *i.e.*, $ofs \leq \text{max_unsigned} - Z(\text{chk})$.

² The function $Z(\text{chk})$ maps memory chunks *byte*, *halfword*, *word* and *doublewords* to 1, 2, 4, and 8, respectively.



4. The interval $[ofs, hi_ofs)$ is within the range of mr .

Otherwise, $check_mem$ returns the null pointer `Vnullptr`.

```

check_mem(perm, chk, addr, MRs) = if  $\exists!$   $mr \in MRs, b.$ 
  let  $ofs = addr - mr.start$  and  $hi\_ofs = ofs + Z(chk)$  in
     $(mr.perm \geq p) \wedge (ofs \% Z(chk) == 0) \wedge$ 
     $(ofs \leq max\_signed - Z(chk)) \wedge (0 \leq ofs \wedge hi\_ofs \leq mr.size)$ 
    then   Vptr(b, ofs)
    else   Vnullptr

```

Semantics. The functions $interp$  and $step$  formalize the implementation of our proof model M_p in the Coq proof assistant by defining a monadic interpreter for rBPF. The top-level recursion $interp$ processes a (monotonically decreasing) $fuel$ argument and a state $rbpf_st$, where $fuel$ represents the number of instructions that the interpreter is allowed to execute. The function $step$ processes individual instructions $C[PC]$. MRs and C are read-only. During normal execution, the flag remains f_n and $interp$ branches to the next instruction. If the flag turns to f_t or f_e while processing an instruction, execution stops. For instance, if $fuel$ reaches zero, the flag turns to f_e . We write $rbpf_st.X$ to denote the value of field X in record $rbpf_st$, and $rbpf_st\{X \leftarrow v\}$ for updating it to v .

```

interp =  $\lambda$  fuel rbpf_st.
  if fuel == 0 then [ $\langle ()$ , rbpf_st{f  $\leftarrow$   $f_e$ } $\rangle$ ]
  else
    match step rbpf_st with
    | [ $\langle ()$ , rbpf_st' $\rangle$ ] =>
      if rbpf_st'.f  $\neq$   $f_n$  then [ $\langle ()$ , rbpf_st' $\rangle$ ]
      else
        interp (fuel-1) rbpf_st'{PC  $\leftarrow$  PC+1}
    |  $\emptyset$  =>  $\emptyset$ 

step =  $\lambda$  rbpf_st.
  match rbpf_st.C[rbpf_st.PC] with
  | Neg dst => [ $\langle ()$ , rbpf_st{R[dst]  $\leftarrow$   $\neg$  rbpf_st.R[dst]} $\rangle$ ]
  | Alu32 op dst src =>
    if check_alu(op, rbpf_st.BR[src]) then
      match alu32(op, rbpf_st.R[dst], rbpf_st.R[src]) with
      | [v] => [ $\langle ()$ , rbpf_st{R[dst]  $\leftarrow$  v} $\rangle$ ]

```

```

    |  $\emptyset \Rightarrow \emptyset$ 
  else
    [(( $\emptyset$ ), rbpf_st{f  $\leftarrow$   $f_e$ })]
| Alu64 op dst src =>
  if check_alu(op, rbpf_st.BR[src]) then
    match alu64(op, rbpf_st.R[dst], rbpf_st.R[src]) with
    | [v] => [(( $\emptyset$ ), rbpf_st{R[dst]  $\leftarrow$  v})]
    |  $\emptyset \Rightarrow \emptyset$ 
  else
    [(( $\emptyset$ ), rbpf_st{f  $\leftarrow$   $f_e$ })]
| Ja ofs => [(( $\emptyset$ ), rbpf_st{PC  $\leftarrow$  PC+ofs})]
| Jump c dst src ofs =>
  if cmp(c, rbpf_st.R[dst], rbpf_st.R[src]) then
    [(( $\emptyset$ ), rbpf_st{PC  $\leftarrow$  PC+ofs})]
  else
    [(( $\emptyset$ ), rbpf_st)]
| Load chk dst reg ofs =>
  match check_mem(Readable, chk, rbpf_st.R[reg]+ofs, rbpf_st.MRs) with
  | Vptr(b, ofs) =>
    match load(chk, rbpf_st.M, b, ofs) with
    | [v] => [(( $\emptyset$ ), rbpf_st{BR[dst]  $\leftarrow$  v})]
    |  $\emptyset \Rightarrow \emptyset$ 
  | _ => [(( $\emptyset$ ), rbpf_st{f  $\leftarrow$   $f_e$ })]
| Store chk dst src ofs =>
  match check_mem(Writable, chk, rbpf_st.R[dst]+ofs, rbpf_st.MRs) with
  | Vptr(b, ofs) =>
    match store(chk, rbpf_st.M, b, ofs, rbpf_st.R[src]) with
    | [N] => [(( $\emptyset$ ), rbpf_st{M  $\leftarrow$  N})]
    |  $\emptyset \Rightarrow \emptyset$ 
  | _ => [(( $\emptyset$ ), rbpf_st{f  $\leftarrow$   $f_e$ })]
| Call imm =>
  let f_ptr = bpf_get_call imm in
  if f_ptr == Vnullptr then [(( $\emptyset$ ), rbpf_st{f  $\leftarrow$   $f_e$ })]
  else
    [(( $\emptyset$ ), rbpf_st{R0  $\leftarrow$  exec_function f_ptr})]
| Exit => [(( $\emptyset$ ), rbpf_st{f  $\leftarrow$   $f_t$ })]
| _ => [(( $\emptyset$ ), rbpf_st{f  $\leftarrow$   $f_e$ })]

```

Result \emptyset marks transitions to crash states, which are proved unreachable based on the definitions of the `check_alu` and `check_mem` functions.

The formal semantics of each rBPF instruction is explained as follows:


- *Neg* updates register *dst* with the the negation of its original value.
- For an arithmetic operation `Alu64 op dst src` (or `Alu32 op dst src`), `check_alu` first checks the validity of *op* with source *src*, then evaluates *op* against destination *dst* using `alu64` (or `alu32`), and stores the result *v* in register *dst*.
- Unconditional jump *Ja* always increments the *pc* by *ofs*, and a conditional *Jump* does so when `cmp(c, src, dst)` holds. If the condition is false, the *Jump* instruction has no effect.
- Similarly, the semantics of memory instructions (*Load-Store*) validates memory accesses using the `check_mem` function, ensuring the absence of undefined behaviors. The functions `load` and `store` are CompCert memory operations mentioned in [Section 3.2.3](#).
- The *Call* instruction selects (using `bpf_get_call`) the trusted system API service designated by an immediate number *imm*. It then calls the chosen service if available (*i.e.*, not a null pointer).
- *Exit* finally terminates the program with the flag *f_t*.

For simplicity, we omit the case of immediate *srcs* for `Alu64`, `Alu32`, and `Store`. If the result is \emptyset , so becomes the monadic state (undefined behavior). Our definition of dynamic checking functions, along with well-formedness conditions (see [Section 5.1.4](#)), ensures that this situation never occurs. In case of error, execution terminates with the flag *f_e*.

5.1.4 Proof of Isolation

Our proof model M_p formalizes the semantics of rBPF and is implemented in Gallina. The assessment of its correctness consists of proving two essential properties:

- The well-formedness of the virtual machine’s state, that is, its registers, memory, verifier and state invariants.
- The achievement of isolation, that is, the isolation of all transitions to a crash state \emptyset using runtime safety checks (*e.g.*, `check_mem`), ergo the impossibility of a transition to an undefined behavior.

The register invariant  states that all rBPF registers contain 64-bit integer values. This rules out 32-bit integers, `Vundef` but also pointers and floating-point numbers, for which the `alu` function may be undefined.

Definition 1 (*register_inv*). $\forall rbpf_st, r. \exists lv. rbpf_st.R[r] = \mathbf{Vlong}(lv)$

As expected, the memory consistency invariant \clubsuit is a bit more elaborate. It states that each rBPF memory region mr , in the CompCert memory model, registers 8-bit integer blocks b of memory M , designated by a pointer $mr.ptr$ to the 32-bit physical $mr.start$ address of b , the 32-bit $mr.size$ of b and at least *Readable* permissions $mr.perm$ across $[0, size)$. Finally, every two regions point to disjoint physical address spaces in M (as per CompCert’s memory regions for $mr'.ptr \neq mr.ptr$).


Definition 2 (*memory_inv*). $\forall rbpf_st, mr \in rbpf_st.MRs. \exists b, start, size. s.t.$
 $mr.ptr = \mathbf{Vptr}(b, 0) \wedge Mem.valid_block\ rbpf_st.M\ b \wedge$
 $is_byte_block\ b\ rbpf_st.M \wedge mr.start = \mathbf{Vint}(start) \wedge mr.size = \mathbf{Vint}(size) \wedge$
 $Mem.range_perm\ rbpf_st.M\ b\ 0\ (Int.unsigned\ size)\ Cur\ mr.perm \wedge$
 $mr.perm \geq Readable \wedge (\forall mr' \in rbpf_st.MRs, mr' \neq mr \rightarrow mr'.ptr \neq mr.ptr)$

Linux eBPF has a verifier to statically analyze eBPF programs and only accept those which are free of undefined behaviors. Our CertrBPF’s verifier, introduced in [Section 5.4](#), ensures the weaker invariant given by [Definition 3](#). The invariant stipulates the minimal pre-condition so that the interpreter can safely run a sequence of instructions C with the fixed length C_len . More precisely, the invariant \clubsuit states that each instruction $C[i]$ references source registers within the range $[0, 10]$ and destination registers with the range $[0, 9]$, and that the target of every jump instruction is within the program range *i.e.*, $0 \leq i + ofs + 1 \leq C_len - 1$.

Definition 3 (*verifier_inv*). $\forall rbpf_st, i, ofs. 0 \leq i \leq rbpf_st.C_len - 1 \rightarrow$
 $0 \leq get_dst(rbpf_st.C[i]) \leq 9 \wedge 0 \leq get_src(rbpf_st.C[i]) \leq 10 \wedge$
 $((rbpf_st.C[i] = \mathbf{Ja}\ ofs \vee rbpf_st.C[i] = \mathbf{Jump}\ ____ ofs) \rightarrow$
 $0 \leq i + ofs + 1 \leq rbpf_st.C_len - 1)$


The last invariant \clubsuit connects two pairs declared in the monadic state: the input BPF binary sequence C with the fixed length C_len , and the memory region list MRs with the fixed size mrs_num . It also specifies the range of C_len and mrs_num with the maximum 32-bit unsigned integer value $0xffffffff$.

Definition 4 (*state_inv*). $\forall rbpf_st.$
 $length(rbpf_st.C) = rbpf_st.C_len \wedge length(rbpf_st.MRs) = rbpf_st.mrs_num \wedge$
 $rbpf_st.C_len \leq unsigned_int32_max \wedge rbpf_st.mrs_num \leq unsigned_int32_max$

These four invariants collectively represent well-formedness as proposed in [Chapter 4](#). Therefore, the following Coq Theorem `sem_preserve_inv`  proves [Theorem 5](#) and states that well-formedness is preserved by the `interp` function.

```
Theorem sem_preserve_inv:  $\forall$  (st st': rbpf_state) (fuel: nat)
(Hinv: register_inv st  $\wedge$  memory_inv st  $\wedge$  verifier_inv st  $\wedge$  state_inv st)
(Hsem: interp fuel st = [(tt, st')]),
register_inv st'  $\wedge$  memory_inv st'  $\wedge$  verifier_inv st'  $\wedge$  state_inv st'.
```

Proof. We first do proof by induction on `fuel`, then prove a collection of lemmas to show each sub-function of `interp` preserves the invariants. The core function `step` performs case analysis on rBPF instructions. In terms of rBPF memory instructions, we prove that `check_mem` doesn't change the global state, implying the preservation lemma of `check_mem`. \square

Similarly, Theorem `interp_no_undef`  proves [Theorem 6](#) and indicates that the dynamic checks of the model M_p are sufficient to ensure the absence of error. In particular, all memory accesses are valid and performed within the dedicated memory regions.

```
Theorem interp_no_undef:  $\forall$  (st: rbpf_state) (fuel: nat)
(Hinv: register_inv st  $\wedge$  memory_inv st  $\wedge$  verifier_inv st  $\wedge$  state_inv st),
interp fuel st  $\neq$   $\emptyset$ .
```

Proof. We used the same proof techniques used in `sem_preserve_inv`: induction on `fuel`, followed by case analysis on each instruction. \square

As a result, our model ensures memory isolation, that is, our virtual machine provides a formal guarantee to prevent faults of itself from affecting other components of the hosted OS. The corollary of `sem_preserve_inv` and `interp_no_undef` is that our virtual machine, obtained by refinement of the proof model, will always use the defensive memory checking semantics (*i.e.*, `check_mem`) to successfully isolate code from other memory regions of the operating system and never crash it.

5.2 A Synthesis-Oriented rBPF Interpreter

The aforementioned model is used to formally describe the abstract behaviours of rBPF and prove its isolation property. However, this proof model is not fit to extract an

executable C implementation: it lacks necessary optimizations and also has a different coding style compared to the original RIOT rBPF implementation in C. Consequently, the resulting code is of low performance and hard to manually validate that the extracted version matches the original one.

This section addresses these issues by introducing a synthesis model. First, it refines M_p into an optimized, safe, and behaviorally equivalent monadic model M_s (Section 5.2.1). Then, it transforms M_s into an effectful C implementation using ∂x (Section 5.2.2).

5.2.1 Synthesis Model

M_s \rightsquigarrow is a refinement of our proof model M_p , following the principle “make M_s as close as possible to the expected target C code”.

Function-level. For each function present in the original C rBPF implementation, there exists a corresponding Gallina function in M_s . These functions have similar arguments, with most of them having the same name and type.

For instance, the C function `_alu64` in the original rBPF implementation, which interprets all rBPF `alu64` instructions, takes an rBPF `opcode` and two pointers to source (`src`) and destination (`dst`) registers. It returns an rBPF flag with type `int`.

```
static int _alu64(uint8_t opcode, uint64_t *src, uint64_t *dst)
```

In the corresponding monadic Gallina specification in M_s , named `step_opcode_alu64`, the arguments include `op` representing the 8-bit rBPF opcode (represented as `nat` for the purpose of proof simplification), `src64` and `dst64` denoting the values of the source and destination registers, an explicit parameter `dst` for updating the destination register, and an implicit parameter `state` for the global monadic state. The function returns the Coq `unit` type since the rBPF flag is a field of the monadic state.

```
Definition step_opcode_alu64 (op: nat) (src64: val) (dst64: val) (dst: reg): M state
  ↪ unit :=
```

Expression-level. Most C expressions intuitively correspond to Gallina structures, for instance, both have the same `if` expression. We highlight two special cases:

- *recursion*: M_s adopts (tail)-recursive Gallina functions to express loops in C.

For instance, the C function `_check_mem` of the original rBPF interpreter, corresponding to `check_mem` in Gallina, implements memory checking defined in [item 5.1.3](#). It takes an rBPF state `bpf` (its type `bpf_t` related to `state` in Gallina), the `size` of memory chunk (corresponding to `chunk`), the memory address `addr`, and the permission `type` (`perm` in Gallina). `_check_mem` uses a for loop to traverse user-declared memory regions with type of *linked list*.

```
static int _check_mem(const bpf_t *bpf, uint8_t size, const intptr_t addr, uint8_t
↪ type) {
    for (const bpf_mem_region_t *region = &bpf->stack_region; region; region =
↪ region->next) {
        ...
    }
}
```

The corresponding Gallina specification `check_mem` first reads the size of memory regions `num` and the memory region list `mrs`³ from the global monadic state, then calls an auxiliary function `check_mem_aux` to recursively access each memory region in the list, where the notation `struct num` is to tell Coq which argument decreases along the recursive calls.

```
Fixpoint check_mem_aux (num: nat) (perm: permission) (chunk: memory_chunk) (addr:
↪ val) (mrs: MyMemRegionsType) {struct num}: M state val :=
    match num with
    | 0 => returnM Vnullptr
    | S n =>
        ...
        check_mem_aux n perm chunk addr mrs
    end.

Definition check_mem (perm: permission) (chunk: memory_chunk) (addr: val): M state
↪ val :=
    do mem_reg_num <- eval_mrs_num;
    do mrs          <- eval_mrs_regions;
    do check_ptr   <- check_mem_aux mem_reg_num perm chunk addr mrs;
    ...
```

3. `MyMemRegionsType` is declared as a Coq list with type of `memory_region`

- *pattern-matching*: M_s defines a C enumerated type as a basic inductive type where the constructors have no arguments, and the *match-with* based on this kind inductive type corresponds to a C *switch-case* structure.

For example, after an opcode masking operation, `_alu64` cases analysis and interprets each rBPF alu64 instruction by a C *switch-case* statement.

```
#define BPF_INSTRUCTION_ALU_OP_MASK    0xf0
#define BPF_INSTRUCTION_ALU_ADD      0x00
#define BPF_INSTRUCTION_ALU_SUB      0x10
static int _alu64(uint8_t opcode, uint64_t *src, uint64_t *dst) {
    uint8_t instruction = opcode & BPF_INSTRUCTION_ALU_OP_MASK;
    switch (instruction) {
        case BPF_INSTRUCTION_ALU_ADD:
            *dst += *src;
            break;
        case BPF_INSTRUCTION_ALU_SUB:
            *dst -= *src;
            break;
        ...
    }
}
```

The related Gallina specification first defines a basic inductive type `opcode_alu64` to describe all cases of rBPF alu64 instructions, *e.g.*, `op_BPF_ADD64` in Gallina for representing `BPF_INSTRUCTION_ALU_ADD` in C. The Gallina model also defines a masking function `byte_to_opcode_alu64` and its monadic version `get_opcode_alu64` to mapping 8-bit opcode, represented by `op` with type `nat`, into `opcode_alu64`. Then the corresponding function `step_opcode_alu64` performs the same (monadic) opcode masking behavior and completes case analysis by a *match-with* statement in Gallina.

```
Inductive opcode_alu64: Type :=
| op_BPF_ADD64
| op_BPF_SUB64
| op_BPF_MUL64
...
Definition byte_to_opcode_alu64 (op: nat): opcode_alu64 :=
match Nat.land op 0xf0 with (**r masking operation **)
| 0x00 => op_BPF_ADD64
```

```

| 0x10 => op_BPF_SUB64
| 0x20 => op_BPF_MUL64
...
Definition get_opcode_alu64 (op: nat): M state opcode_alu64 :=
returnM (byte_to_opcode_alu64 op).

Definition step_opcode_alu64 (op: nat) (src64: val) (dst64: val) (dst: reg): M state
↪ unit :=
do opcode_alu64 <- get_opcode_alu64 op;
  match opcode_alu64 with
  | op_BPF_ADD64 =>
    upd_reg dst (Val.add1 dst64 src64)
  | op_BPF_SUB64 =>
    upd_reg dst (Val.sub1 dst64 src64)
  | op_BPF_MUL64 =>
    upd_reg dst (Val.mull dst64 src64)
  ...

```

As a consequence, this principle aims to capture all possible optimization strategies of the original design and enhance readability.

One of the optimizations is opcode masking, which utilizes the instruction classes of the rBPF encoding mentioned in Table 3.1, to quickly decode rBPF instructions. The opcode masking optimization consists of two-level masking operations: it firstly classifies the class of the given instruction using the first-level masking operation and then identifies the specific instruction using the second-level masking. The C function `_instruction` interprets single rBPF instruction, it firstly applies a mask (line 8) to the opcode with `BPF_INSTRUCTION_CLS_MASK`, *i.e.*, `0x07` (line 1). For example, the masked opcode of rBPF alu64 is `BPF_INSTRUCTION_CLS_ALU64` (line 9) with a value of `0x07` (line 4). The second-level masking operation of rBPF alu64 is defined in `_alu64`, which masks the opcode with `BPF_INSTRUCTION_ALU_OP_MASK` to map rBPF ADD64 to `BPF_INSTRUCTION_ALU_ADD`, rBPF SUB64 to `BPF_INSTRUCTION_ALU_SUB`, and so on.

```

1  #define BPF_INSTRUCTION_CLS_MASK      0x07
2  #define BPF_INSTRUCTION_CLS_LD       0x00
3  ...
4  #define BPF_INSTRUCTION_CLS_ALU64    0x07
5  static int _instruction(bpf_t *bpf, uint64_t *regmap, const bpf_instruction_t **pc){
6  ...

```

```

7   switch (instruction->opcode & BPF_INSTRUCTION_CLS_MASK) {
8     case BPF_INSTRUCTION_CLS_ALU64:
9       return _alu64(instruction->opcode, src, dst);
10    ...
    
```

The Gallina function related to `_instruction` is `step`. It calls a monadic function `get_opcode` to implement the first-level opcode masking and then `step_opcode_alu64` performs the second-level masking operation by invoking `get_opcode_alu64`, a monadic function implemented using the `returnM` operator and the `byte_to_opcode_alu64` pure Gallina function that completes the opcode masking and maps masked opcodes to the inductive type `opcode_alu64`.

```

Definition step: M state unit := ...
do opc <- get_opcode op;
match opc with
| op_BPF_ALU64 => ...
  step_opcode_alu64 ...
    
```

$M_p = M_s$. Both M_p and M_s use the same monadic state $st : rbpf_state$ as in [Section 5.1](#). Hence, the simulation relation $R \subseteq rbpf_state \times rbpf_state$, required by [Theorem 7](#), is equality. As a result, we prove the stronger result \clubsuit that both $interp : nat \rightarrow M\ unit$, the M_p interpreter, and $interp_dx : nat \rightarrow M\ unit$, the M_s interpreter, denote the exact same function.

```

Theorem equivalence_between_proof_model_and_synthesis_model:
   $\forall$  (st: rbpf_state) (fuel: nat),
    interp fuel st = interp_synthesis fuel st.
    
```

Proof. The key proof is to show the the synthesis model with the opcode masking is identical to the proof model. Since rBPF opcodes are 8-bit, *i.e.*, within the range $[0, 255]$, a straightforward proof approach is to perform a case analysis on all possible opcodes and establish that two models are behaviourally equivalent. To simplify the proof, we represent rBPF opcodes as natural numbers in Coq. We define a collection of special rBPF instructions in Coq to represent undefined opcodes, such as those greater than 255 or those lacking a corresponding rBPF meaning, *e.g.*, `0xd4` represents a byteswap instruction in Linux eBPF but is not supported by rBPF. \square

5.2.2 C-ready Model

This section elaborates on the process of extracting a C implementation from the synthesis model M_s using ∂x . ∂x accepts monadic Gallina programs that utilize ‘identifiable’ inductive types, and M_s meets this *monadic* requirement as it includes the option state monad M along with two standard monad operators ($bindM$ and $returnM$). M_s involves not only the default ∂x ‘identifiable’ inductive types nat and $bool$ that are respectively translated into `unsigned int` and `_Bool` in C, but also rBPF-specific types *e.g.*, the *register* type and CompCert *value* types. To enable ∂x to identify these types, we first provide a reusable ∂x -CompCert library that defines a mapping relation from CompCert Value types and functions to C types and operators. We then declare the remaining rBPF-related types, *e.g.*, *reg*, as corresponding to `unsigned int`.



∂x -CompCert Library . The mapping from CompCert *value* to C types is not bijective, as a CompCert *value* is either a 32-bit machine integer `Vint`, a 64-bit machine integer `Vlong`, or a pointer `Vptr`, etc. The case of `Vptr` is particularly delicate, as the target type contextually relies on bit-size and signedness. To sort this out, we define a ∂x -CompCert library: It firstly renames the *val* type and the *int* type to match the correct C types, as shown in Table 5.1. For example, `val64_t`, `valu32_t`, `vals32_t` are *Val* types mapped to `unsigned long long`, `unsigned int` and `int`, respectively. Then it maps CompCert constructs and constant functions to C operators and constants, *e.g.*, ‘`Val.addl`’ to ‘+’, and ‘`true`’ to ‘1’, etc. The library also specifies mappings for other CompCert types, for instance, mapping the CompCert memory chunk *memory_chunk* into `unsigned int` in C, where the constructor `Mint32` corresponds to `4U`.

Table 5.1 – Mapping relation in the ∂x -CompCert library

	CompCert	C
Types	<code>valu32_t/vals32_t/valptr8_t ...</code> <code>sint32_t/uint32_t ...</code>	<code>unsigned int/int/unsigned char* ...</code> <code>int/unsigned int ...</code>
Constructions	<code>Int.repr(-2)/Vzero ...</code>	<code>1/-2/0 ...</code>
Constants	<code>Val.addl/subl/mull ...</code>	<code>+/-/* ...</code>

$M_s = M_{dx}$. The refinement from M_s into a C-ready model M_{dx} is a renaming process,  where all CompCert types are renamed into ∂x identifiable types according to the ∂x -CompCert library.

The equivalence proof \Downarrow between the synthesis model M_s and the C-ready model M_{dx} is trivial as the additional ∂x configuration of M_{dx} has no effect on the semantics.

Theorem `equivalence_between_synthesis_model_and_dx_model`:

```

 $\forall$  (st: rbpf_state) (fuel: nat),
  interp_synthesis fuel st = interp_dx fuel st.

```

Proof. This proof is straightforward because M_{dx} is simply M_s with renamed types. The only operation is to unfold each renamed type into its original form using the Coq `unfold` tactic. \square

Code Extraction with ∂x . We use ∂x to generate the final C implementation from M_{dx} . The extracted C implementation preserves the structure of the original Gallina code, and the extracted C functions directly operate on actual memory locations as CompCert memory operations map to C expressions with a dereference.

For instance, consider the `step_mem_st_reg` function that interprets rBPF store with register instructions. Its arguments include the value of the source register `src` (*i.e.*, a 64-bit CompCert value, `val64_t`), the memory address `addr` (*i.e.*, a 32-bit address, `valu32_t`), and the opcode `op` (*i.e.*, an 8-bit opcode, `nat8`). `step_mem_st_reg` performs the second-level opcode masking⁴ using `get_opcode_mem_st_reg`, one of the masked opcodes being `op_BPF_STXW`. This opcode only stores the low 32 bits of `src` into the address `addr_ptr` if memory checking is validated, *i.e.*, the return value of `check_mem` is not null pointer. When `check_mem` returns null, as checked by `eq_ptr_null`, the function `step_mem_st_reg` reports a memory error message using `upd_flag`.

Definition `step_mem_st_reg` (src: val64_t) (addr: valu32_t) (op: nat8): M unit :=

```

do opcode_st <- get_opcode_mem_st_reg op;
match opcode_st with
| op_BPF_STXW =>
  do addr_ptr <- check_mem Writable Mint32 addr;
  if eq_ptr_null addr_ptr then
    upd_flag BPF_ILLEGAL_MEM
  else (** i.e. Mem.storev Mint32 addr_ptr src *)
    store_mem_reg Mint32 addr_ptr src
...

```

4. `step` completes the first-level masking

Coq’s `nat8`, a renaming of `nat`, is mapped to `unsigned char`. The opcode `op_BPF_STXW` is translated to the decimal representation ‘99’, and its hexadecimal format `0x63` represents the masked opcode for the rBPF store 4 bytes with register instruction. The error message `ILLEGAL_MEM` is mapped into ‘-2’, following the original rBPF C implementation, and the permission `Writable` is converted into ‘2U’. The constant function `eq_ptr_null` is translated into an operation to check whether a pointer is null. The ‘`match opcode_st with`’ construct is extracted to ‘`switch (opcode_st) case`’. The C functions `step_mem_st_reg`, `check_mem` and `store_mem_reg` all include an additional monadic argument `st`.

```
void step_mem_st_reg(struct rbpf_state* st, unsigned long long src, unsigned int
↪ addr, unsigned char op){
    unsigned char opcode_st;
    unsigned char *addr_ptr;
    opcode_st = get_opcode_mem_st_reg(op);
    switch (opcode_st) {
        case 99:
            addr_ptr = check_mem(st, 2U, 4U, addr);
            if (addr_ptr == 0) {
                upd_flag(st, -2);
            } else { // i.e. *(unsigned int *) addr_ptr = src
                store_mem_reg(st, 4U, addr_ptr, src);
            }
        ...
    }
}
```

5.3 Simulation Proof of the C rBPF Virtual Machine

In this section, we explain how to establish [Theorem 8](#) for the Clight code of our virtual machine, which is derived from ∂x , and compiled into a Clight AST in Coq using `CLIGHTGEN`.

Clight State. The rBPF state of the monadic model M_{dx} , as defined in [Section 5.1.2](#), is implemented as a record type in Gallina (see [Figure 5.4](#) left). ∂x translates this global state into a global variable named `bpf_state` in C.

```
struct bpf_state {
    int pc_loc;
```

```

int bpf_flag;
unsigned long long regs_st[11];
unsigned int mrs_num;
struct memory_region *bpf_mrs;
unsigned int ins_len;
const unsigned long long * ins;
};

```

Therefore, the corresponding Clight memory, translated by CLIGHTGEN, contains three additional blocks to represent the other fields of the Gallina state. *state_block* denotes the global state (`struct bpf_state * st`), *mrs_block* represents the memory regions (`struct memory_region * bpf_mrs`), and *ins_block* stands for the input rBPF binary (`const unsigned long long * ins`). The layout and content of those blocks are depicted in Figure 5.4.

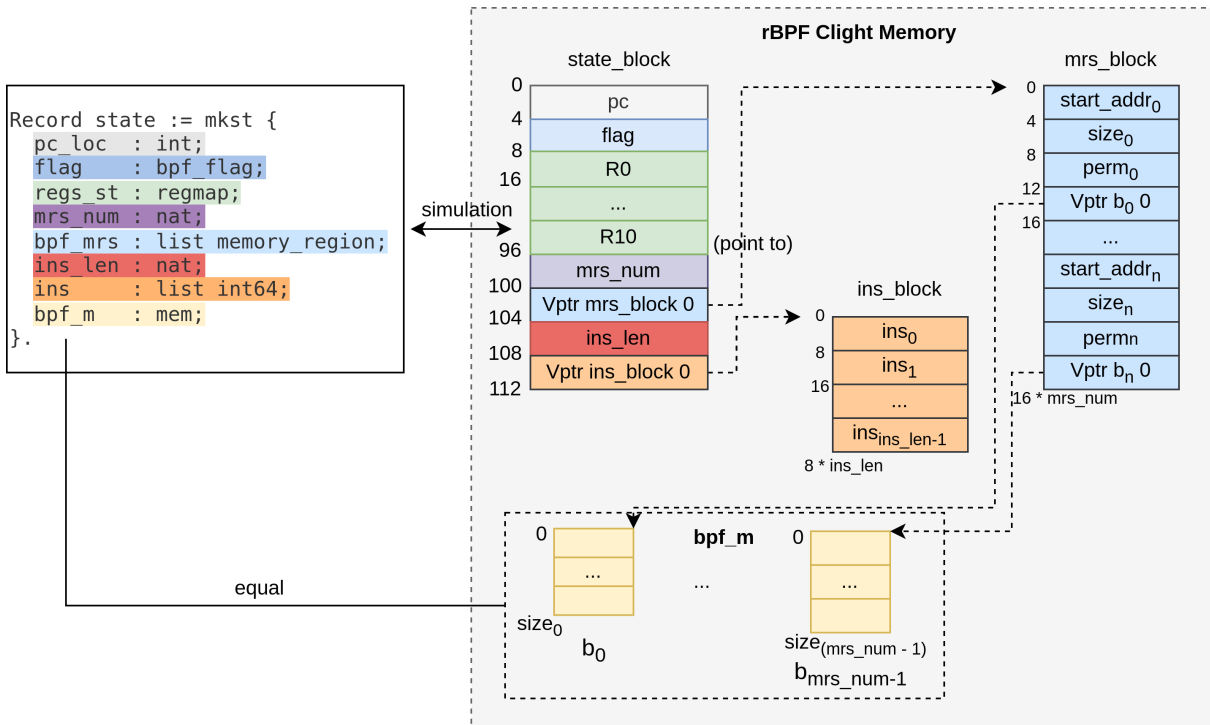


Figure 5.4 – Simulation relation R between st_{rbpf} , left, and `rBPFClight`, right.

Simulation Relation \mathcal{R} . A crucial ingredient of Theorem 8 is the simulation relation between the Gallina state monad and the Clight state which is essentially made of a CompCert memory. The Gallina state comprises a CompCert memory that models the


various memory regions available to the rBPF program. This memory may also contain other blocks that are not modified by the virtual machine but represent other kernel data-structures. The simulation relation stipulates that such blocks also exist in the Clight memory and have the same content, as shown in [Figure 5.4](#).

Solid arrows in [Figure 5.4](#) are simulation relations between *state_block* and *rbpf_st*. Solid lines are the equalities between the rBPF memory *m* and blocks in rBPFClight memory. Dashed lines indicate relations of pointers to blocks in CompCert memory. The encoding exploits the fact that each field of the Gallina state has a known length. Thus, every field can be encoded as a continuous sub-block. As a result, the program counter is obtained from the first 4 bytes: loading a memory chunk of type *Mint32* at offset 0 retrieves the *pc* field of the Gallina state. The next 4 bytes encode the enumerated type flag. Here, each constructor of type flag is assigned an integer. The next 11×64 bits are used to encode the register bank of the Gallina state.

$$Rs(rbpf_st, m, state_block, ins_block, mrs_block) =$$

$$\bigwedge \begin{cases} rbpf_st.pc & = \text{load } Mint32 \ m_{clight} \ state_block \ 0 \\ rbpf_st.flag & = \text{load } Mint32 \ m_{clight} \ state_block \ 4 \\ rbpf_st.R[R0] & = \text{load } Mint64 \ m_{clight} \ state_block \ 8 \\ \dots & \end{cases}$$

The next elements of the Clight block represent the lists of instructions and of memory regions. In a functional language, lists are potentially of unbounded length and have a polymorphic type. Here, our lists always have fixed lengths and elements of fixed size. As a result, a list is directly encoded by a field specifying its length followed by a pointer to its memory block. The elements of the list are stored continuously in the pointed block.

Systematic Proof of Simulation. Since the ∂x tool is syntax-directed, there is a systematic correspondence between the source Gallina and the target C code. We exploit this property to design a minimal Clight logic geared toward our simulation proof. Our *Clightlogic*  generalizes the translation validation theorem ([Theorem 8](#)) to accommodate Gallina functions and C functions with multiple arguments. In that case, we have a precondition that states that the Gallina and C arguments are linked pairwise by a refinement relation. Most of the arguments are numeric values and, in this case, the refinement relation states that the Gallina and C values are the same. The *Clightlogic* also provides a syntax-directed proof principle for each pair of Gallina/C syntactic construct.

For instance, the $bindM$ operator translates to a sequence in the C code. Also, the result of a Gallina function call is bound to a local variable in C. Moreover, the local variable v below stands for the monadic state in C and points to the state memory block.

$$\partial x(bindM f (\lambda x.g)) = (vx = f_C(v); g_C(v, vx))$$

To exploit this pattern, our invariants take the form of an association list mapping each local variable to a set of C values that is obtained by partially evaluating a refinement relation with the Gallina value computed by the function (Figure 5.4). To evaluate f , one needs to have a refinement relation Rs between the Gallina state st and the C value of v in memory m . Now, suppose that $fst = [r, st']$. Since f_C is a correct refinement of f , relations $Rs(st', v, m')$ and $Rr(r, x)$ hold for the value x of the local variable vx in the current environment. We conclude by mapping $vx \mapsto Rr r$ and use this invariant to refine g by g_C .

The translation validation theorem \clubsuit proves a forward simulation relation from Coq to Clight. A backward simulation relation can be constructed as Gallina programs are functions and Clight is *determinate*.

5.4 CertrBPF Verifier

Linux eBPF’s compiler and runtime system do not enforce type or memory safety. Instead, safety is verified prior to execution using a static analyzer that checks programs validity. As both the size and complexity cannot fit the requirements of an MCU architecture, CertrBPF instead provides a simple (linear time) but formally verified verifier, CertrBPF-verifier \clubsuit , which ensures the invariant $verifier_inv$ (Definition 3). Accordingly, it scans an input rBPF program (*i.e.*, a list of 64-bit bytecode instructions) and rejects it when:

1. a source destination register is greater than 10, and a target destination register is greater than 9,
2. the offset of a jump instruction is out of the instruction sequence bounds,
3. or the last instruction is not the *Exit* instruction (opcode 0x95).

Static verification of these properties allows the interpreter to skip unnecessary dynamic checks. Our verifier adopts the same end-to-end verification method as the interpreter, Chapter 4.

The virtual machine state in CertrBPF-verifier is a strict subset of the interpreter's state \clubsuit : $verifier_state ::= (C, M)$ consists of a sequence of instructions C and a memory M .

The proof model of CertrBPF-verifier is implemented a monadic function `verifier` : $M \text{ bool}$. When `verifier` returns *true*, it establishes a priori the harmlessness of the BPF program. Otherwise, it represents `verifier` detects some violations of the aforementioned three rules and in this case, the CertrBPF interpreter cannot be enabled.

```

Theorem verifier_well_formedness_and_safety :
   $\forall$  (st: verifier_state) (b: bool),
    verifier st = [(b , st)].

Theorem verifier_imply_inv :
   $\forall$  (st: verifier_state) (st': rbpf_state) (Hinclude: st  $\subset$  st')
    (Hpre : verifier st = [(true, st)]),
    verifier_inv st'.

```

Theorem *verifier_well_formedness_and_safety* \clubsuit intuitively denotes that the verifier state st is always unchanged whatever the result returns. It proves both [Theorem 5](#) and [Theorem 6](#): the former holds due to the unchanged state and the latter holds because `verifier` always returns a result (no crash occurs). In summary, the verifier has the following properties:

- no assumption (every state is well-formed);
- never crashes (safety);
- never modifies the VM state.

In addition, the Coq theorem *verifier_imply_inv* \clubsuit states that if the *verifier* returns *true*, *verifier_inv* holds.

Considering that the verifier's proof and synthesis models are exactly the same, the simulation relation $R_v \subseteq verifier_state \times verifier_state$ required by [Theorem 7](#) is equality \clubsuit . The C-ready model of CertrBPF-verifier is refined by reusing the ∂x -CompCert library and then it is translated into a C version by ∂x . Last, CertrBPF-verifier reuses the *Clightlogic* to prove the simulation proof between its proof model and the C implementation \clubsuit .

5.5 Optimization

The previous section introduced all existing optimizations of the original rBPF, which are also implemented in the synthesis model M_s . In this section, we discuss a novel optimization to accelerate the memory-checking process.

5.5.1 `check_mem` Optimization

The formalization of the rBPF interpreter uses the native `check_mem` function, which is not optimized. For a given rBPF memory instruction, the `check_mem` function iterates over all memory regions, in sequence, to find if the instruction points to a valid memory region. While such implementation is correct, performance can be increased by changing the order in which `check_mem` iterates over the memory regions.

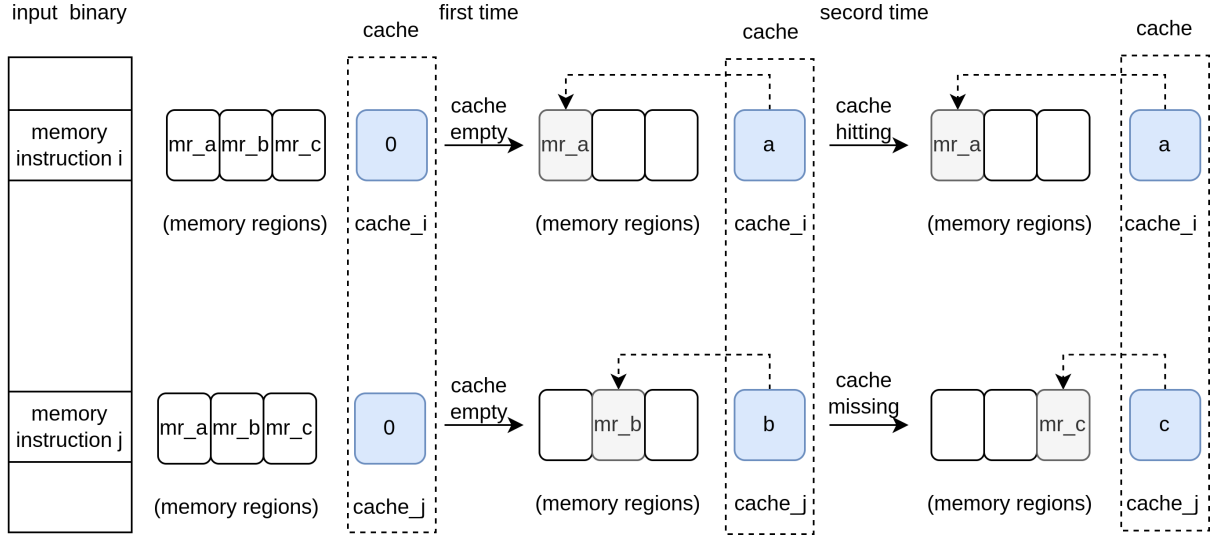
In this section, we explore an alternative implementation of the `check_mem` function as a candidate for runtime optimization. When an rBPF memory instruction is executed for the first time, `check_mem` loops in sequence over the memory regions until the accessed memory is reached, as the non-optimized `check_mem` does. The accessed memory region is stored in a cache, so that when the same instruction is called for a second time, the `check_mem` first looks at the cache before iterating over all the other memory regions.

High Level Intuition. Two scenarios for the optimized `check_mem` function are graphically depicted in [Figure 5.5](#). We provide a detailed explanation for each scenario.

The input binary, on the left of [Figure 5.5](#), contains two memory instructions, at locations i and j . The list of memory regions is displayed next to the input binary array, and duplicated for readability (although, in practice, only one array of memory regions exists). The cache, in dotted line, is initialized with null 0 elements.

When a memory instruction of the input binary is interpreted, the `check_mem` function is called. Since the cache is initially empty, the optimized `check_mem` behaves, on the first call, as the non-optimized `check_mem` function, iterating in sequence over the memory regions to find if the memory instruction is valid. When the function identifies the valid memory region (*e.g.*, mr_a for the i -th instruction and mr_b for the j -th instruction), the cache is updated with the index of that memory region (*e.g.*, a for the i -th instruction and b for the j -th instruction).

The second time that an instruction is interpreted, the optimized `check_mem` function behaves differently from the non-optimized `check_mem` version. It first checks the memory

Figure 5.5 – Synthesis model: `check_mem` optimization

region pointed to by the cache. If that memory region is valid for the instruction, the cache keeps the reference, resulting in a *cache hit*. If not, it sequentially checks all other memory regions until a valid one is found. The cache is then updated with the new valid memory region (*e.g.*, the j -th memory instruction, on the second time, belongs to the memory region mr_c).

Optimized `check_mem` Function. We fix $perm$, chk , mrs , $cache$, $addr$, and pc , to respectively denote a [CompCert permission](#), a [CompCert memory chunk](#) (size of memory block to access), a list of memory regions, a cache where elements of the list are indexes in the list of memory regions, a memory address, and a program counter. We also write $l[n \mapsto v]$ for updating the n -th element of list l with value v .

The new optimized `check_mem` ([Algorithm 1](#)) operates as follows: It first checks if the cache is empty ([line 2](#)), if the cache is not empty, the algorithm translates the memory address $addr$ to a pointer based on the history memory region indexed by $cache[pc]$ ([line 3](#)). If the result is valid which represents cache hitting, then the algorithm directly returns *true* along with the unchanged cache. The other two cases, when the cache is empty ([line 10](#)) and when the cache is missing ([line 5](#)), follow a similar pattern. They calculate the number of memory regions and iterate over all memory regions using the recursive function `check_all_mrs` ([line 12](#)).

The iteration proceeds from right to left, and `check_all_mrs` skips the region $cache_id$

during the iteration (line 16). If no valid pointers are found for all memory regions, the function `check_all_mrs` returns `false` along with the old `cache` (line 14). Otherwise it returns `true` along with the updated `cache` containing the new memory region (line 22). The function `check_one_mem_region` is used to verify whether the input address `addr` satisfies the condition that the memory interval $[addr, addr+chk)$ falls within the range of the `id`-th memory region. It also checks the alignment and permission of the interval. It's important to note that in this simplified algorithm, we treat `check_one_mem_region` as a boolean function, In the full specification (see item 5.1.3), `check_one_mem_region` should return a valid pointer.

Implementation. The Coq development reuses the CertrBPF Gallina specification. The optimized rBPF interpreter `✂` takes a new state, lifting the existing CertrBPF state with an additional field named `cache`. We stipulate:

- `cache` represents a list of caches, with the same length as the input rBPF binary list, *i.e.*, each memory instruction in the binary list corresponds to a cache in the `cache` list.
- The value of each cell of `cache` should be within the range $[0, mrs_num]$ where `mrs_num` is the number of memory regions. Initially, each `cache` cell is empty (0 by default). When it is updated with a `cache_id`, this cache points to the corresponding memory region with index `cache_id - 1`.

Our new interpreter implementation introduces a flag '`opt_flag : bool`': when users set the flag to `true`, the optimization is enabled. If `opt_flag = false`, our specification is equivalent to the formally verified CertrBPF interpreter.

$$bpf_interp (opt_flag : bool) (fuel : nat) (ctx_ptr : val) : M\ val.$$

Benefiting from the workflow described earlier, we can automatically extract an executable C program from our Gallina model.

5.5.2 Equivalence Proof

The monadic definition of the rBPF interpreter is designed such that every sub-function of the interpreter operates as a monadic function defined over the same state. The benefit of having a monadic model is that the binding operator of the monad composes the value and threads the state over each function. For instance, given $f_1 : A \rightarrow M\ B$

Algorithm 1: The `check_mem` optimization algorithm

Data: ($perm : permission$), ($chk : memory_chunk$), ($mrs : list\ memory_region$), ($cache : list\ nat$), ($addr : val$), ($pc : int$)

Result: ($is_valid : bool$), ($new_cache : list\ nat$)

```

1 check_mem :
2 if  $cache[pc] \neq null$  then
3   |  $is\_valid \leftarrow check\_one\_mem\_region(mrs[cache[pc]], perm, chk, addr)$  ;
4   | if  $\neg is\_valid$  then
5     | return  $check\_all\_mrs(size(mrs), id, perm, chk, mrs, cache, addr, pc)$ ;
6     | /* cache missing */
7   | else
8     | return ( $true, cache$ ) ; /* cache hitting */
9   | end
10 else
11 | return  $check\_all\_mrs(size(mrs), 0, perm, chk, mrs, cache, addr, pc)$  ;
12 | /* cache empty */
13 end

```

Data: ($n : nat$), ($cache_id : nat$), ($perm : permission$), ($chk : memory_chunk$), ($mrs : list\ memory_region$), ($cache : list\ nat$), ($addr : val$), ($pc : int$)

Result: ($is_valid : bool$), ($new_cache : list\ nat$)

```

12 check_all_mrs :
13 if  $n = 0$  then
14 | return ( $false, cache$ );
15 else if  $num = cache\_id$  then
16 | return  $check\_all\_mrs(n - 1, cache\_id, perm, chk, mrs, cache, addr, pc)$ ;
17 else
18 |  $is\_valid \leftarrow check\_one\_mem\_region(mrs[n], perm, chk, addr)$  ;
19 | if  $\neg is\_valid$  then
20 | | return  $check\_all\_mrs(n - 1, cache\_id, perm, chk, mrs, cache, addr, pc)$ ;
21 | else
22 | | return ( $true, cache[pc \mapsto n]$ ); /* cache updating */
23 | end
24 end

```

and $f_2 : B \rightarrow M C$, then the composition is simply the binding of f_1 and f_2 , written as $do\ x \leftarrow f_1; f_2$. The drawback, however, is that every function is defined over the same option-state monad with a global state. As a consequence, if the global state changes (*e.g.*, due to an optimization), all invariants have to be re-verified.

Alternatively, each function of the model can be defined over the subset of the state that it modifies. For instance, if a function f_1 only modifies the sub-state s of the state *state*, the modified version f'_1 has the new signature $f'_1 : A \times s \rightarrow (B \times s)$. The benefit of such approach is that if the global state *state* is modified (*e.g.*, due to an optimization), but the modification does not affect s , then the invariants of f'_1 still hold without requiring additional proof. The drawback, however, is that composition of f'_1 and f'_2 (for some projection of f_2) is no longer as straightforward as the monadic binding, as the signatures may not coincide. We refer to this transformation as *simplification*.

Our strategy is then the following. We retain the monadic model for design purposes, as monadic composition simplifies specification. We apply the *simplification* transformation to the rBPF interpreter with optimization. We then prove the correctness of the *check_mem* optimization. Finally, we demonstrate the correctness of the simplification through an equivalence proof.

Challenge

Following the workflow of CertrBPF, our new proof model adopts the monadic form with an option *state* monad M . The standard refinement proof adopted by CompCert and CertrBPF is to prove the theorem *optimization_correctness* that the two models (non-optimized and optimized) preserve a proper forward simulation relation.

The simulation relation $match_states \subseteq state \times state$ is straightforward: the equality between all other fields in two states, except for the *cache*.

$$match_states(st_1, st_2) \stackrel{def}{=} \bigwedge \begin{cases} st_1.pc = st_2.pc \\ st_1.flag = st_2.flag \\ \dots \end{cases}$$

```

Theorem optimization_correctness: ...
  (Hsim: match_states st1 st2)
  (Hinterp: interp false fuel st1 = [(res, st1')]),
  ∃ st2',
  
```

```

interp true fuel st2 = [(res, st2')] /\
match_states st1' st2'.

```

Theorem *optimization_correctness* only considers the case when the monadic interpreter returns successfully because the existing isolation proof [Yua+22] guarantees the monadic CertrBPF interpreter (*i.e.*, *opt_flag = false*) never crashes.

We show the challenge that it is quite difficult to directly prove the optimization is correct based on our monadic model. We explain our solution in the next section.

We first introduce the function tree of our Gallina model, as the proof process of the theorem follows this tree structure. As depicted in Figure 5.6, left is the monadic model with the global state, right is our simplified model, and the grey region includes the shared functions whose simulation proofs can be eliminated on the right side. We highlight four nodes of the function tree:

- `bpf_interp`: The top function of our new CertrBPF model.
- `step`: It interprets a single rBPF instruction with an initial state, resulting in a new state.
- `upd_reg`: The leaf node updating the register map field of the global monad state.
- `check_mem`: The key function, enabling the *check_mem* optimization when *opt_flag* is *true*. We underline that the optimized case and the non-optimized one return different states, as the former can modify the cache field of the global state.

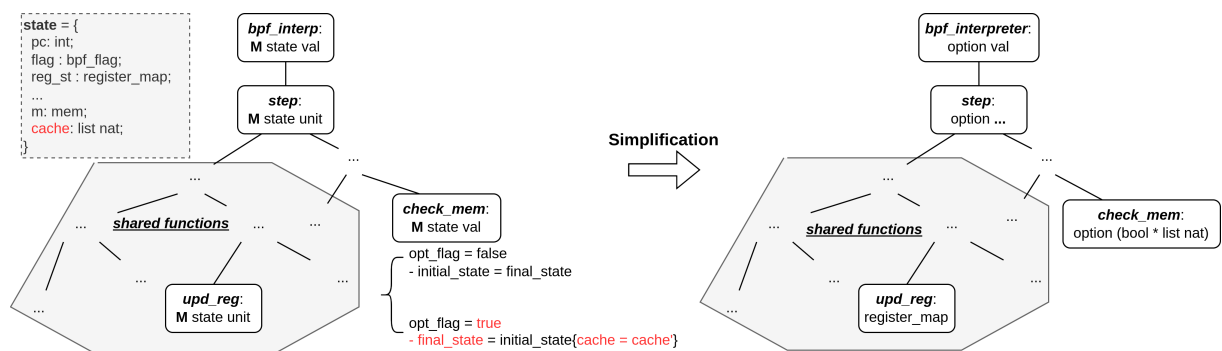


Figure 5.6 – Function tree of the optimized interpreter

To complete this standard state-based refinement proof, it requires a forward simulation proof of each node in Figure 5.6. The main challenge arises because all monadic functions in the tree share the same global state, and `check_mem` has different effects on

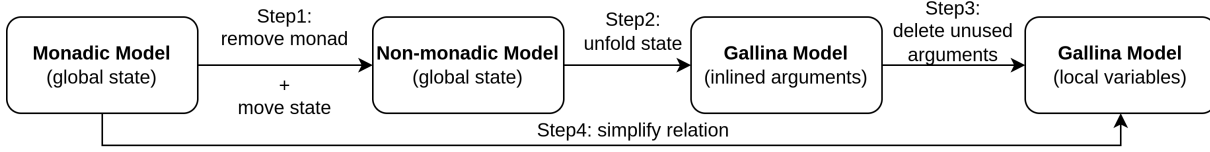


Figure 5.7 – The simplification process.

this state, depending on the optimization flag *opt_flag*. The [existing experience](#) from CertrBPF has shown evidence that this approach is quite complex and requires extensive effort to prove numerous, albeit trivial, detailed lemmas.

Our solution is direct: we forget the monadic structure and replace the global state with proper inline arguments using a so-called *simplification* process. This results in a significant simplification of the final theorem’s proof, allowing us to omit the simulation proof of most shared functions, as they have exactly the same behavior in both the optimized and non-optimized models.

Simplification

As depicted in [Figure 5.7](#), the *simplification* process consists of four steps:

$$\begin{aligned}
 & f (_ : a) : M b \\
 (\text{step1}) & \Rightarrow f_1 (_ : a) (_ : state) : option (b \times state) \\
 (\text{step2}) & \Rightarrow f_2 (_ : a) (_ : t_1) (_ : t_2) \dots (_ : t_n) : option (b \times t_1 \times t_2 \times \dots \times t_n) \\
 (\text{step3}) & \Rightarrow f_3 (_ : a) (_ : t_j) (_ : t_j) \dots : option (b \times t_i \times t_j \times \dots) \\
 (\text{step4}) & (f, f^{opt}) \in R \Rightarrow (f_3, f_3^{opt}) \in R_{simpl}
 \end{aligned}$$

Step 1: Removing the Monad and Monad Operations. We begin by unfolding our option state monad M along with its operations. Subsequently, we move the initial state to an argument for enhanced readability: This syntactic transformation doesn’t modify the semantics of Gallina programs.

Step 2: Replacing the Global State with Inline Arguments. In this step, we assume that the global *state* contains several fields with the signature $t_1 \times t_2 \times \dots \times t_n$, we construct a new function by:

1. Unfolding the initial state as a list of arguments with types t_1, t_2, \dots, t_n , and

2. Replacing the final state with the projection of all its components.

Step 3: Bottom-Up Deletion of Unused Arguments and Outputs. According to the function tree of our monadic model, we systematically remove all unused arguments and reduce fields of outputs starting from leaf nodes. Only modified components allow to be output and their types are reserved in the type signature.

For instance, the initial monadic functions (see comments below) and the final simplified versions of `upd_reg`, `step`, and `bpf_interpreter` are shown as follows:

- `upd_reg` only modifies the register map `regmap` field in the global state, therefore we only reserve this type in the input and output.
- `step` requires most fields of the monadic state, except for the bpf flag, as `step` is only executed when the interpreter status is normal. The `step` function returns a new program counter (affected by `Branch` instructions), a new register map (modified by most instructions, e.g., ALU), a new CompCert memory (due to `Store` instructions), a bpf flag (resulting from operations like division by zero, etc.), and a new cache (updated by the `check_mem` optimization). Some fields, such as the binary instruction list `l`, the input binary size `len`, and the memory region list `mrs`, are not returned, indicating they remain unchanged by `step`.
- `bpf_interpreter` 🐛 takes all components of the global state as parameters and returns the final result if successful.

```

(**r Definition upd_reg (r: reg) (v: val) : M state unit := ... *)
Definition upd_reg (r: reg) (v: val) (rs: regmap): regmap := ...

(**r Definition step (opt_flag: bool): M state unit := ... *)
Definition step (opt_flag: bool) (pc: int) (cache: list nat) (l: list int64)
  (len: nat) (rs: regmap) (mrs_num: nat) (mrs: MyMemRegionsType) (m: mem):
  option (int * regmap * mem * bpf_flag * list nat) := ...
match ins with
| Alu ... => match step_alu ... with
  | [] => []
  | [(rs', f)] => [(pc, rs', m, f, cache)]
end
...
(**r Definition bpf_interpreter (fuel: nat) (opt_flag: bool): M state val :=
  ↪ ... *)
Definition bpf_interpreter (opt_flag: bool) (fuel: nat) (pc: int) (cache: list nat)

```

```
(l: list int64) (len: nat) (rs: regmap) (mrs_num: nat) (mrs: MyMemRegionsType)
(m: mem): option (val * int * regmap * mem * bpf_flag * list nat) := ...
```

Step 4: Simplify the Simulation Relation. In the simplified model, since there are no states, the simulation relation R can be equivalently replaced by a much simpler and more intuitive input-output relation R_{simpl} : the simulation relation between a pair of initial states of functions f and f^{opt} is transformed into an input relation. This input relation specifies that the simplified functions f_3 and f_3^{opt} must have the same input values for simplified arguments. The simulation of the final state is expressed as the relation that f_3 and f_3^{opt} yield the same value for simplified output fields.

For example, the lemma `step_preserves_simulation_relation` 🐛 declares an input-output relation derived from `match_states`.

- *Input:* The initial simulation relation is replaced by the constraint that all inline input arguments (pc, register map, CompCert memory, and bpf flag) must be identical for the optimized function (`opt_flag = true`) and the non-optimized one.
- *Output:* The final simulation relation is expressed as two parts:
 - *Explication:* All fields that are used in `match_states` and also exist in the output should be identical, *e.g.*, the new pc value pc' .
 - *Implication:* All fields that do not appear in the output are unmodified, *e.g.*, the read-only memory region list mrs .

```
Lemma step_preserves_simulation_relation: ...
  (Hstep: step false pc cache l len rs mrs_num mrs m = [(pc', rs', m', f', cache')],
   ∃ cache1,
   step true pc cache l len rs mrs_num mrs m = [(pc', rs', m', f', cache1)]).
```


Last, the construction of the final state is replaced by finding a new *cache*.

Proof

This section mainly discusses two essential theorems:


- *The correctness of simplification (Theorem 9):* It aims to prove the equivalence between the initial monadic model and the simplified model.

- *The correctness of the `check_mem` optimization (Theorem 10):* This theorem is dedicated to demonstrating the equivalence between the non-optimized model and the optimized model.

Theorem 9 (Simplification Correctness ). *Assume that the monadic interpreter `interp` takes the initial state `st1` and successfully produces the result `res` and the final state `st2`, the simplified version `bpf_interpreter`, which accepts all fields of `st1` as arguments, returns the same result.*

```
Theorem simplification_correctness: ...
(Hinterp: interp opt_flag fuel st1 = [(res, st2)]),
bpf_interpreter opt_flag fuel (pc_loc st1) (cache st1) (ins st1) (ins_len st1)
(regs_st st1) (mrs_num st1) (bpf_mrs st1) (bpf_m st1) =
  [(res, pc_loc st2, regs_st st2, bpf_m st2, flag st2, cache st2)].
```

Proof. The key aspect to note is that the monadic `check_mem` function may potentially modify the global state. Therefore an associated lemma is proved to illustrate that this function has no impact on all other fields utilized by all subsequent monadic functions. \square

Theorem 10 (Optimization Correctness ). *Assuming that the simplified rBPF interpreter accepts the same arguments, the non-optimized model and the optimized model produce identical results when the `check_mem` optimization is disabled or enabled.*

```
Theorem optimization_correctness_simpl: ...
(Hmem_disjoint: memory_regions_disjoint mrs_num mrs m0)
(Hcache_inv: cache_inv cache l mrs_num)
(Hinterp: bpf_interpreter false ... = [(res, pc, rs, m, f, cache)]),
 $\exists$  cache1,
  bpf_interpreter true ... = [(res, pc, rs, m, f, cache1)].
```

This theorem requires two assumptions:

- The user-declared memory regions `mrs` are disjoint, signifying that there is no overlap between any two memory regions. This assumption is directly derived from the memory invariant of the original isolation proof.
- Due to the introduction of the new field `cache`, an additional invariant is used to formalize the stipulation discussed in Section 5.5.1 (see the implementation part) for proving Theorem 10: `cache_inv` specifies both the length of the cache list and the valid range of each element within the list.

Proof. We first case analysis on rBPF instructions, the proof of the non-memory instructions is trivial because both non-optimized and optimized models execute identical behaviors. The memory instruction cases are non-trivial because the `check_mem` function of two models has different behaviors: the optimized model may update `cache`. Therefore we prove an important lemma `check_mem_preserves_simulation_relation` that indicates two models return the same result pointer `ptr` but different `caches`.

```
Lemma check_mem_preserves_simulation_relation: ...
  (Hcheck: check_mem false mrs_num ... = Some (ptr, cache')),
  ∃ cache1,
  check_mem true mrs_num ... = Some (ptr, cache1).
```

Next, the `check_mem` lemma proof consists of three cases of the cache in the optimized model:

- *cache not exists* (`cache_id = 0`): This is the most simplest case which represents the cache is empty and the optimized model performs the normal memory checking as same as the non-optimized version, the only difference is that once it finds a valid pointer, it updates the cache with the corresponding memory region index before output. In this case, the proof is trivial.
- *cache exists* (`cache_id ≠ 0`): The optimized model observes the cache is not empty, there are two cases,
 - *cache missing*: The input address is not valid in the corresponding memory region of `cache_id`, therefore the optimized model performs the normal memory checking as same as the non-optimized version but skips the memory region `cache_id` (Algorithm 1: line 16-17). This case proves by induction on the number of memory regions `mrs_num` and returns a new cache.
 - *cache hitting*: The proof is also by induction on `mrs_num`, and this case doesn't modify the cache because the input address is valid in the corresponding memory region of `cache_id`. The proof requires that the non-optimized version is also (and only) valid in this memory region where we use the assumption that memory regions are disjoint.

□

5.6 Conclusion

In this chapter, we have applied the refinement methodology proposed in [Chapter 4](#) to generate a verified C implementation of rBPF, the implementation of BPF hosted by the RIOT operating system, from a Gallina specification in Coq. All the refinement steps have been mechanically verified using the Coq proof assistant to minimize the TCB. We prove that our virtual machine never crashes and it doesn't produce any runtime errors.

CERTBPF-JIT

Linux eBPF by default is a 64-bit machine architecture because it is derived from the 64-bit RISC-V family and all eBPF registers are 64-bits. Its variant, rBPF, follows the same design. As rBPF is typically deployed in 32-bit IoT devices, the 64-bit design naturally leads to inefficiency. There are additional expenses to simulate 64-bit rBPF registers and operators on low-power 32-bit microcontrollers.

Furthermore, rBPF employs an interpreter as its execution engine, which tends to be slow. Linux eBPF reduces this issue by introducing a runtime component named JIT (Just-In-Time) compiler to optimize the execution speed of eBPF programs. The JIT compiler performs a binary translation by compiling source bytecodes to native target machine code. When considering the incorporation of JIT compilation into rBPF, a significant challenge arises. rBPF includes runtime defensive checks for its instructions, such as `check_mem`, which are designed to capture all memory undefined behaviors. Consequently, implementing JIT compilation in rBPF becomes more complex and error-prone.

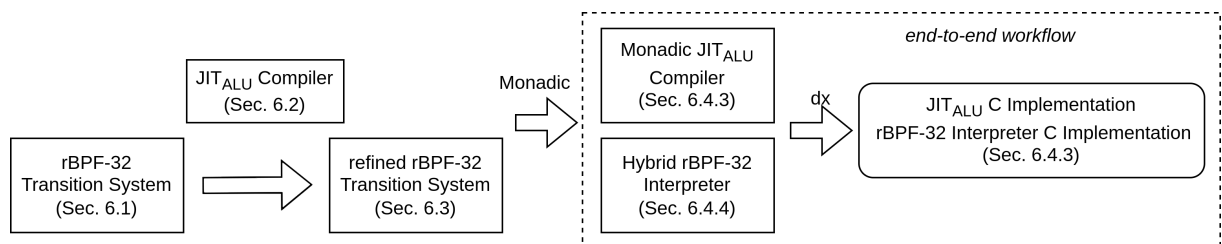


Figure 6.1 – Chapter Structure: CertrBPF-JIT

To address these issues, we first propose rBPF-32: a 32-bit version of rBPF specifically designed for IoT applications, and formalize its inductive semantics in Gallina ([Section 6.1](#)). We then introduce a JIT design for rBPF-32 that translates rBPF-32 `Alu32` instructions into target machine code ([Section 6.2](#)). Consequently, we refine rBPF-32 into a new version, called rBPF-32-JIT whose semantics executes both rBPF bytecode and `jited` machine code ([Section 6.3](#)). We present HAVM (hybridly accelerated virtual

machine): a hybrid design that combines our JIT compiler with a 32-bit CertrBPF interpreter. We demonstrate how to reuse our verification workflow to extract an executed JIT compiler in C and compose a completed HAVM (hybridly accelerated virtual machine) C implementation (Section 6.4). Last, we discuss the proof details of HAVM, outlining the next steps in our journey (Section 6.5).

6.1 rBPF-32

To bridge the gap between the 64-bit rBPF VM and 32-bit target platforms, we introduce a 32-bit version of rBPF, named rBPF-32. In rBPF-32, registers are 32-bits in size, and all 64-bit operators *e.g.*, `alu64` have been removed.

In contrast to the monadic semantics mentioned in Section 5.1, rBPF-32 adopts the standard CompCert transition systems to describe the rBPF program behaviours. This choice allows us to reuse the CompCert backend semantics to express the behaviours of `jited` code generated by our JIT compiler.

Following the CompCert definition style, the state of rBPF-32 is represented as a pair $rbpf32_state ::= (R, M)$, consisting of a CompCert memory model M and the register state R , which associates (32-bit) values with the rBPF-32 registers. In particular, R includes an additional program counter register PC , in comparison to the original CertrBPF (Figure 5.2).

The syntax of rBPF-32, as shown in Figure 6.2, closely resembles that of CertrBPF. It differs in two key aspects: i) None of the operators are 64-bit, and ii) The *doublewords* memory chunk (*i.e.*, 8 bytes) is deprecated.

Transition Semantics of rBPF-32 🐞. The operational semantics for rBPF-32 is defined as a label transition system. The transition relation, denoted as $G, C, MR \vdash st \xrightarrow{t} st'$, represents one execution step from state st to state st' in the global CompCert environment G , rBPF binary code C , and memory region mapping relation MR between $\text{ptr}(b, 0)$ and a 32-bit integer value $addr$. The trace t denotes the observable events generated by this execution step. In our case, only `Call` instructions can generate an associated event in t , others always result in $t = \epsilon$.

The core of rBPF-32's semantics is a transition function $T(ins, st) = [st']$ that determines the new state st' after executing instruction ins in the initial state st . In particular, the program counter `PC` is either incremented for arithmetic, branching instructions

Registers :	
$R ::= R_0 \mid R_1 \mid \dots \mid R_9$	General-purpose
R_{10}	Stack Frame Pointer
PC	Program Counter
Memory Chunk :	
$chk ::= byte \mid halfword \mid word$	1/2/4 Bytes
Instruction :	
$ins ::= \text{Neg32 } dst$	Unary arithmetic
$\text{Alu32 } op \ dst \ src$	Binary arithmetic
$\text{Ja32 } ofs$	Unconditional branch
$\text{Jump32 } cmp \ dst \ src \ ofs$	Conditional branch
$\text{Load32 } chk \ dst \ reg \ ofs$	Memory load
$\text{Store32 } chk \ dst \ src \ ofs$	Memory store
$\text{Call32 } imm$	Call
Exit	Return

Figure 6.2 – Syntax of rBPF-32 instruction set

(when the condition is false), call, and memory instructions, or set to the branch target for branching instructions (when the condition is true).

For the transition rules of arithmetic ($\text{Neg32 } dst$ and $\text{Alu32 } op \ dst \ src$) and branching instructions ($\text{Ja32 } ofs$ and $\text{Jump32 } cmp \ dst \ src \ ofs$), the first two premises model abstractly the act of reading and decoding the n -th instruction ins , which is pointed to by the program counter PC , from the list C . Then, the rule executes ins and returns a new state. The detailed definition of T is omitted here as it closely resembles the monadic functions mentioned in [Section 5.1.3](#).

$$(ALU + Jump) \frac{R[PC] = \text{Vint}(n) \quad C[n] = [ins] \quad T(ins, (R, M)) = [(R', M')]}{G, C, MR \vdash (R, M) \xrightarrow{\epsilon} (R', M')}$$

The transition rule of memory instructions are somewhat complicated. This complexity arises from the fact that the original memory address is a 32-bit integer (*i.e.*, $R[reg] + ofs = \text{int}(v)$) while the CompCert Memory M requires a pointer (*i.e.*, $\text{ptr}(b, ofs')$). To bridge this gap, the rule leverages the global rBPF memory region mapping relation

MR to translate the expected block b into the corresponding 32-bit integer start address $start_addr$ where v can be represented as $\text{int}(v) = start_addr + ofs'$ for a new offset ofs' . Intuitively, MR serves as an implementation of the transformation function of $check_mem$ (item 5.1.3), and the CompCert memory `load/store` mechanism performs most types of constraint checks of $check_mem$.

- *Permission*: CompCert memory operators verify the permission order using the function `Mem.perm` before accessing any memory content.
- *Alignment*: CompCert invokes the function `align_chunk` to check if the offset is aligned.
- *No Overflow*: CompCert models an infinite memory so itself doesn't explicitly check for memory address overflow. An additional $MAX(ofs) = ofs \leq max_unsigned - Z(chk)$ function is used to perform this check.
- *Bounded Checks*: Typically, when the transition functions `exec_load` and `exec_store` return successfully, it implies that CompCert has performed the bounded check of the given rBPF-32 memory instruction. If a CompCert memory operation accesses out-of-bound regions, the decode and encode functions of CompCert memory return an undefined value $Vundef$, which leads to the failure of the transition functions.

$$\begin{array}{c}
 R[PC] = \mathbf{Vint}(n) \quad C[n] = [\mathbf{Load32} \ ck \ dst \ reg \ ofs] \quad R[reg] + ofs = start_addr + ofs' \\
 (\text{Load}) \frac{MR(b) = start_addr \quad MAX(ofs') \quad exec_load(ck, dst, \mathbf{Vptr}(b, ofs'), (R, M)) = [(R', M')]}{G, C, MR \vdash (R, M) \xrightarrow{c} (R', M')}
 \end{array}$$

$$\begin{array}{c}
 R[PC] = \mathbf{Vint}(n) \quad C[n] = [\mathbf{Store32} \ ck \ dst \ src \ ofs] \quad R[reg] + ofs = start_addr + ofs' \\
 (\text{Store}) \frac{MR(b) = start_addr \quad MAX(ofs') \quad exec_store(ck, src, \mathbf{Vptr}(b, ofs'), (R, M)) = [(R', M')]}{G, C, MR \vdash (R, M) \xrightarrow{c} (R', M')}
 \end{array}$$

$$\begin{array}{c}
 R[PC] = \mathbf{Vint}(n) \quad C[n] = [\mathbf{Call32} \ imm] \quad G(imm) = [\mathbf{External}(ef)] \\
 (\text{Call}) \frac{external_call(ef, \overrightarrow{args}, M) \xrightarrow{t} (v, M') \quad R' = R\{PC \leftarrow R[PC] + 1, R_0 \leftarrow v\}}{G, C, MR \vdash (R, M) \xrightarrow{t} (R', M')}
 \end{array}$$

The last rule describes the big-step execution of an rBPF-32 call instruction, which invokes an external function (*i.e.*, RIOT-OS system calls). Following the standard BPF calling-convention, the predicate `external_call` models the BPF call behaviours using the semantics of CompCert external functions, introduced in [Section 3.2.2](#): it passes the first five arguments ($\overrightarrow{args} = [R[R_1]; R[R_2]; R[R_3]; R[R_4]; R[R_5]]$) from the registers R_1, \dots, R_5 along with the external function pointer declared in the global environment. The returned result is then stored in register R_0 , and PC moves to the next instruction.

rBPF-32 also defines two predicates:

- `initial(M, Ptr, st)`: The state st is an initial state where the CompCert memory is specified by M , and the register state consists of: R_{10} points to the tail of rBPF stack Ptr , and all other registers have default values of 0.

$$\begin{aligned} \text{initial}(M, Ptr, st) = st &= (R, M) \wedge \\ R &= \{R_0 \leftarrow \text{Vint}(0), R_1 \leftarrow \text{Vint}(0), \dots, R_{10} \leftarrow Ptr, PC \leftarrow \text{Vint}(0)\} \end{aligned}$$

- `final(C, Ptr, st, res)`: The state st is a final state for the rBPF program C where the register satisfies that R_0 has the returned value res , R_{10} stores the static address of the tail of rBPF stack Ptr , and PC points to the `Exit` instruction in the list C .

$$\begin{aligned} \text{final}(C, Ptr, st, res) = st &= (R, M) \wedge \\ R &= \{R_0 \leftarrow \text{Vint}(res), \dots, R_{10} \leftarrow Ptr, PC \leftarrow \text{Vint}(n)\} \wedge \\ C[n] &= [\text{Exit}] \end{aligned}$$

6.2 Just-In-Time Compilation

Our JIT compiler, named `JITALU` 🐛, is designed to only translate rBPF-32 `Alu32` instructions into target binary code. `Non-Alu32` instructions are still interpreted by the host rBPF-32 virtual machine (introduced in [Section 6.4](#)). This choice is made because the semantics of rBPF-32 memory instructions involves several dynamic memory-safety checks, which are naturally easier to implement through an interpreter rather than in a hypothetically faster list of non-trivial `jited` ARM instructions. In addition, this document only selects ARM as our target architecture, particularly ARM 32-bit, which currently dominates the semiconductor architecture industry. For instance, ARMv7-M with microcontroller profile is a popular 32-bit processor which is usually deployed on IoT devices.

Section 6.2.1 first presents the structure of our JIT compiler, followed by Section 6.2.2, which introduces the core mapping from rBPF-32 A1u32 to ARM binary, and Section 6.2.3, which discusses the interaction between rBPF-32 binary and ARM binary.

6.2.1 Structure

Figure 6.3 illustrates the JIT translation process from a list of rBPF-32 A1u32 binary instructions to ARM binary. The former uses the little-endian encoding shown in Figure 3.1 while the latter adopts the THUMB encoding. The procedure consists of seven stages (second line from the top):

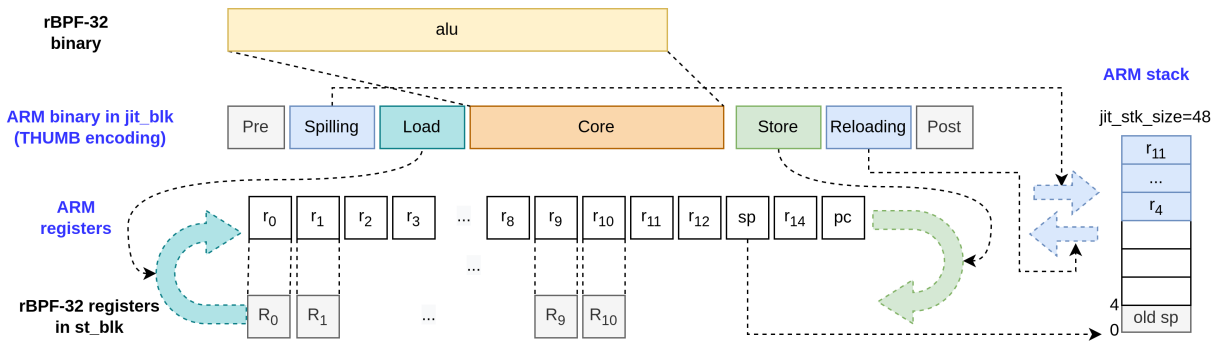


Figure 6.3 – JIT procedure: from rBPF-32 A1u32 binary to ARM binary

- The *Pre* stage saves the start address of a special CompCert block *st_blk*.
- *Spilling* copies ARM registers into the stack (arrow right).
- *Load* transfers rBPF-32 registers' value into ARM registers (bend arrow: left).
- *Core* executes the *jited* binary code on ARM registers.
- *Store* updates rBPF-32 registers with the values from ARM registers (bend arrow: right).
- *Reloading* moves stack slots into ARM registers (arrow left).
- *Post* modifies the stack pointer *sp* and the program counter *pc*.

As shown in Figure 6.3, JIT_{ALU} assumes there is an allocated stack frame with a fixed size of 48 bytes. The old *sp* is stored at the beginning of the new stack frame.

Additional CompCert Blocks. From the perspective of the ARM binary level, the *jited* binary code and the rBPF-32 register map should be stored in some memory blocks. Therefore, our JIT compiler involves two specific CompCert blocks:

- *jit_blk*: stores all `jited` binary code. Its memory layout is a list of ARM binary with a pre-allocated size.
- *st_blk*: records an rBPF-32 register map at a specific position within this block. Its memory layout is somewhat complicated: similar to Figure 5.4, we stipulate that the (part) layout of *st_blk* corresponds to Figure 6.4¹.

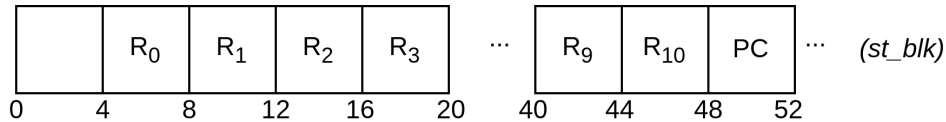


Figure 6.4 – state block layout (The first 52 byte)

ARM Register Usage. Each ARM register plays a specific role in JIT_{ALU} :

- $r_0 - r_{10}$ are associated with the rBPF-32 register map $R_0 - R_{10}$.
- r_{11} serves as a temporary register because it is an ARM callee-save register.
- r_{12} records the start address of *st_blk*.
- $r_{13}(sp)$ always points to the beginning of the currently allocated stack frame.
- $r_{14}(ra)$ stores the return address for branching to the correct location after executing `jited` code.
- $r_{15}(pc)$ is the ARM program counter.

JIT Calling-Convention. JIT_{ALU} assumes that before calling the `jited` code, the initial state of the ARM register map is as follows:

- r_1 contains the start address of the global state.
- *sp* points to the bottom of the allocated stack frame.
- the link register *i.e.*, r_{14} holds the return address.
- *pc* points to the first `jited` instruction of the given rBPF-32 A1u32 binary.
- all other registers retain their previous value.

We start by introducing the kernel mapping rules of the *Core* stage, then explain the roles of other stages step by step.

1. the first four bytes will be used later, see Section 6.5.2.

6.2.2 Core Mapping

JIT_{ALU} directly translates rBPF-32 bytecode into ARM binary. For simplicity, all mapping rules are described using pseudo-code with formal semantics. *e.g.*, ‘ $\text{ADD } R_d R_s$ ’ (*i.e.*, $\text{Alu32 add } R_d R_s$) represents an rBPF-32 register-related addition instruction, with its formal semantics derived from [Section 6.1](#). It intuitively denotes the addition of the values in rBPF-32 registers R_d and R_s , and the result is written to R_d . The ARM semantics is borrowed from the verified CompCert ARM backend, *e.g.*, ‘ $\text{add } r_d r_d r_n$ ’ represents the same behaviour based on the ARM registers.

Core Stage. The rBPF-32 Alu32 instructions can be categorized into those with the last argument as a register or an immediate number. We first discuss the mapping rules from the former case to the CompCert ARM binary.

As shown in [Table 6.1](#), the mapping rules are straightforward: from an rBPF-32 binary operation (*e.g.*, add) to its corresponding ARM instruction (*e.g.*, ADD), the only distinction concerns target patterns. The first five instructions share the same pattern: the second operand is a register. The MOV and MUL instructions follow a similar pattern to the first five instructions.

[Table 6.1](#) does not translate rBPF-32 shift and division instructions into ARM instructions because those rBPF-32 instructions require additional dynamic checks to avoid *shift-out-of-range* and *div-by-zero* issues. These translations can be performed using special techniques, which we will discuss in [Section 6.5.2](#). The mapping rules also exclude the rBPF-32 MOD instruction because CompCert lacks explicit syntax and semantics for *mod* (see [here](#)).

Table 6.1 – Mapping relation from rBPF-32 Alu32 reg to ARM

rBPF-32 Alu32 instruction	ARM instructions
$\text{ADD}/\text{SUB}/\text{OR}/\text{AND}/\text{XOR } R_d R_s$	$\text{add}/\text{sub}/\text{orr}/\text{and}/\text{eor } r_d, r_d, r_s$
$\text{MOV } R_d R_s$	$\text{mov } r_d, r_s$
$\text{MUL } R_d R_s$	$\text{mul } r_d, r_d, r_s$
$\text{LSH}/\text{RSH}/\text{ARSH}/\text{DIV}/\text{MOD } R_d R_s$	error

For the rBPF-32 immediate instructions, the mapping rules are more complex because ARM instructions can accept immediate numbers in different ranges.

For the first five instructions in [Table 6.2](#), if the immediate constant is in the range $[0, 255]$, each instruction is directly mapped to an 8-bit-immediate ARM instruction. If

Table 6.2 – Mapping relation from rBPF-32 Alu32 imm to ARM

rBPF-32 Alu32 instruction	ARM instructions
<i>ADD/SUB/OR/AND/XOR</i> $R_d\ i$ ($0 \leq i \leq 255$) ($256 \leq i \leq 65535$)	<i>add/sub/orr/and/eor</i> r_d, r_d, i
($65536 \leq i$)	<i>movw</i> r_{11}, i ; <i>add/sub/orr/and/eor</i> r_d, r_d, r_{11}
	<i>movw</i> $r_{11}, lo_16(i)$; <i>movt</i> $r_{11}, hi_16(i)$; <i>add/sub/orr/and/eor</i> r_d, r_d, r_{11}
<i>MOV</i> $R_d\ i$ ($0 \leq i \leq 65535$) ($65536 \leq i$)	<i>movw</i> r_d, i
	<i>movw</i> $r_d, lo_16(i)$; <i>movt</i> $r_d, hi_16(i)$
<i>MUL</i> $R_d\ i$ ($0 \leq i \leq 65535$) ($65536 \leq i$)	<i>movw</i> r_{11}, i ; <i>mul</i> r_d, r_d, r_{11}
	<i>movw</i> $r_{11}, lo_16(i)$; <i>movt</i> $r_{11}, hi_16(i)$; <i>mul</i> r_d, r_d, r_{11}
<i>LSH/RSH/ARSH</i> $R_d\ i$ ($0 \leq i \leq 31$)	<i>movw</i> r_{11}, i ; <i>lsl/lsr/asr</i> r_d, r_d, r_{11}
<i>DIV</i> $R_d\ i$ ($i \neq 0$)	<i>movw</i> r_{11}, i ; <i>udiv</i> r_d, r_d, r_{11}
<i>MOD</i> $R_d\ i$	<i>error</i>

the constant falls within the range $[256, 65535]$, the constant is first copied into ARM register r_{11} using a *movw* instruction, and then mapped to an ARM instruction with r_{11} as the second operand. For constants greater than or equal to 65536, the high 16 bits are loaded into r_{11} , followed by the low 16 bits, before performing the operation. For instance, ‘*ADD* $R_0\ 0xff$ ’ is translated into ‘*add* $r_0\ r_0\ \#0xff$ ’ while ‘*ADD* $R_0\ 0xffff$ ’ is mapped into three ARM instructions ‘*movw* $r_{11}\ \#0xffff$; *movt* $r_{11}\ \#0xf$; *add* $r_0\ r_0\ r_{11}$ ’.

For the rBPF-32 *MOV* instruction, the mapping rules generate a *movw* instruction if the immediate constant is less than 16 bits, or similarly generate an extra *movt* instruction to move the high 16 bits of the supplied constant. For the other instructions, we have two options: either moving the constant into r_{11} and generating a related (register) instruction, as is the case for *MUL* and *DIV*, or ensuring that the input rBPF-32 binary list never includes such instructions, as is the case for *MOD*.

Finally, given a list of rBPF-32 Alu32 binary L , we construct a list of ARM binary L'_{core} using the mapping rules of Table 6.1 and Table 6.2. The tail of L'_{core} also includes two ARM binary instructions to update the rBPF-32 PC using the length of L . This ensures that during the jited execution, our rBPF-32 interpreter can skip all subsequent *alu* instructions and locate the correct instruction for the next execution.

$$L'_{core} = [\dots; \text{movw } r_{11}, \text{length}(L); \text{str } r_{11}, [r_{12}, \#0]]$$

6.2.3 Interaction

In the Core stage, source instructions operate over rBPF-32 registers, while the `jited` ARM code operates on ARM registers. Hence, a consistent interaction of register-level transfers is mandatory: copying rBPF-32 registers into ARM before the Core stage, and updating the rBPF-32 registers after the Core stage. It is additionally essential to preserve the ARM calling convention as the rBPF-32 interpreter ‘calls’ the `jited` code. This issue is dealt with during the Spilling and Reloading stages.

Load and Store Stages. We perform a dataflow analysis to compute i) the minimum set LD of rBPF-32 registers that need to be loaded into ARM registers before L'_{core} ; ii) and the minimum set ST of ARM registers loaded into rBPF-32 registers after L'_{core} . The simplified version of our dataflow analysis is as follows: It takes two initial sets, LD and ST , and updates them with registers according to the input rBPF-32 instruction.

$$dataflow_analysis(ins, LD, ST) = \begin{cases} (LD, ST \cup \{R_d\}) & \text{if } ins = MOV\ R_d\ i \\ (LD \cup \{R_s\}, ST \cup \{R_d\}) & \text{if } ins = MOV\ R_d\ R_s \\ (LD \cup \{R_d\}, ST \cup \{R_d\}) & \text{if } ins = ADD\ R_d\ i \mid SUB\ R_d\ i \mid \dots \\ (LD \cup \{R_d, R_s\}, ST \cup \{R_d\}) & \text{if } ins = ADD\ R_d\ R_s \mid SUB\ R_d\ R_s \mid \dots \end{cases}$$

The Load stage includes a list of ARM load instructions $L'_{load} = [ldr\ r_i, r_{12}, \#(i*4+4)]$ for all $r_i \in LD$. The offset is $i*4$ because each rBPF-32 register is 32 bits, and an additional 4 bits are necessary due to the memory layout described in [Figure 6.4](#). Similarly, the Store stage generates $L'_{store} = [str\ r_i, r_{12}, \#(i * 4 + 4)]$ for all $r_i \in ST$.

Spilling and Reloading Stages. The `jited` code should preserve all callee-saved registers ($CSR = \{r_i \mid 4 \leq i \leq 11\}$) to their initial values. This is the responsibility of the Spilling stage, which copies all used callee-saved ARM registers into the ARM stack before the Load stage, and the Reloading stage, which resets all used callee-saved registers into the initial values after the Store stage. The key is to compute all used callee-saved registers, where r_{11} is always included because L'_{core} uses it to update rBPF-32’s PC value.

$$CSR_{used} = (CSR \cap (LD \cup ST)) \cup \{r_{11}\}$$

The Spilling stage consists of a sequence of instructions $L'_{spilling} = [str\ r_i, sp, \#(i*4)]$ for all $r_i \in CSR_{used}$. The offset is $i * 4$ because each ARM register is 32 bits. Conversely,

the Reloading stage is a sequence of the form $L'_{reloading} = [ldr\ r_i,\ sp,\ \#(i * 4)]$ for all $r_i \in CSR_{used}$.

Pre and Post Stages. Considering the Load and Core stages may override r_1 , the Pre stage moves r_1 's information to r_{12} that is not used by any rBPF-32 instructions and is not a callee-save register. $L'_{pre} = [mov\ r_{12},\ r_1]$.

The Post stage frees current stack frame and branches to the return address.

$L'_{post} = [ldr\ sp,\ sp,\ \#0;\ b\ r_{14}]$.

The entire JIT_{ALU} process generates a list of ARM binary L' by using Coq List's concatenation operator '++' to combine the aforementioned lists in a specific sequence:

$L' = L'_{pre} ++ L'_{spilling} ++ L'_{load} ++ L'_{core} ++ L'_{store} ++ L'_{reloading} ++ L'_{post}$.

Example. To illustrate the entire JIT_{ALU} process, let's consider a source rBPF-32 Alu32 snippet composed of three instructions: $[ADD\ R_0\ R_1;\ MOV\ R_5\ R_0;\ MUL\ R_6\ 0xf]$.

Initially, both LD and ST are \emptyset , We sequentially apply each input instruction for the dataflow analysis, resulting in the final state where $LD = \{R_0;\ R_1;\ R_6\}$ and $ST = \{R_0\ R_5\ R_6\}$. Then the used callee-saved registers CSR_{used} are $R_5, R_6,$ and R_{11} .

We can now introduce each stage of the `jited` ARM code generated from the input three instructions. The first stage *Pre* is always `mov r12, r1`. The Spilling stage saves all registers of CSR_{used} in the stack frame, and the Load stage copies three rBPF-32 registers of LD into ARM $r_0, r_1,$ and $r_6,$ respectively. The Core stage includes the corresponding ARM instructions defined in [Table 6.1](#) and [Table 6.2](#), along with two instructions `'movw r11, #3; str r11, r12, #48'` to update the rBPF-32's PC with the length of the input list, *i.e.*, 3. JIT_{ALU} stipulates that rBPF-32's PC register is stored at offset 48, as shown in [Figure 6.4](#). The Store stage updates all modified rBPF-32 registers, *i.e.*, ST , and the Reloading stage resets all used call-save registers to their previous values stored in the stack frame during the Spilling stage. The last stage is *Post*.

```

mov  r12, r1      ;. Pre Stage
str  r5, [sp, #20] ;. Spilling Stage
str  r6, [sp, #24]
str  r11, [sp, #44]
ldr  r0, [r12, #4] ;. Load Stage
ldr  r1, [r12, #8]
ldr  r6, [r12, #28]
add  r0, r0, r1   ;. Core Stage

```

```

mov r5, r0
movw r11, #0xf
mul r6, r6, r11
movw r11, #3      ;. update rBPF-32's PC
str r11, [r12, #48]
str r0, [r12, #4]  ;. Store Stage
str r5, [r12, #24]
str r6, [r12, #28]
ldr r5, [sp, #20] ;. Reloading Stage
ldr r6, [sp, #24]
ldr r11, [sp, #44]
ldr sp, [sp, #0]  ;. Post Stage
b r14

```

6.3 Refinement of rBPF-32: rBPF-32-JIT

We introduce a refined rBPF-32 model to formalize the behaviours of executing both the original rBPF binary script and the new `jited` binary code generated from each *Alu32* segment of the input script by JIT_{ALU} . Intuitively, this refined model behaves:

- *At the rBPF-32 level*: when it finds rBPF-32 non-Alu32 instructions from the input rBPF script, it applies the existing rBPF-32 transition rules.
- *At the ARM level*: when it finds rBPF-32 Alu32 instructions, it calls the CompCert ARM transition functions to interpret the corresponding `jited` binary.

To perform the switch between the rBPF-32 level and the ARM level, the refined model also provides a formal Gallina specification to connect the rBPF-32 transition system to the CompCert ARM semantics.

6.3.1 Symbolic CompCert ARM

Unfortunately, the ARM binary generated by JIT_{ALU} cannot be directly interpreted by the existing CompCert ARM backend, as the latter defines ARM semantics at the assembly level rather than the binary level. To address this issue, we define an ARM decoding function and a symbolic CompCert ARM semantics that employs symbolic execution to preserve the ARM calling convention.

ARM Decode. We implement a decode function in Gallina that translates binary ARM instructions to standard CompCert ARM instructions with assembly syntax. Each input binary instruction is either 16 or 32 bits and adopts different THUMB encoding forms. The details of the THUMB encoding can be found in [ARMv7-M Reference Manual](#).

Symbolic Execution. To ensure that the `jited` code doesn't break the ARM calling convention mechanism, *i.e.*, after completing the execution of the last `jited` code, the final register state should ensure that all ARM callee-save registers are reset to their initial values, *i.e.*, the ARM register values before calling the `jited` code. We adopt a symbolic execution technique to address this concern. In our CompCert variant, the ARM register map `SReg` is symbolic: each register `sr` is either an abstract value or a concrete value binding to an actual ARM register `r`.

$$\text{SReg} \ni sr ::= \text{abstract}(r) \mid \text{concrete}(r)$$

Initially, all ARM registers have their abstract values, *e.g.*, $\text{SReg}[r_0] = \text{abstract}(r_0)$. The concrete values of registers are inserted by `jited` code during the execution.

We define a function `init_state` to create a new ARM environment for interpreting binary code. It first copies values from the arguments list `args` to ARM argument registers ($r_0 - r_3$) of the initial symbolic register map `init_rs`, according to a given function signature `sig`. Then, `init_state` allocates a new memory block `stk` with a fixed stack size `sz` in the CompCert memory. It stores the previous stack pointer `sp` at position `pos` in this allocated block `stk` and updates the stack pointer with the start address of this block (*i.e.*, the bottom). Finally, it stores the return address, *i.e.*, the next address of the old `pc`, to r_{14} . Since we promise the first argument always points to the start address of the `jited` binary code to be executed, `init_state` also assigns the program counter `pc` with the first argument value r_0 .

```

init_state(sig, args, sz, pos, m) =
  match alloc_arguments(sig, args, init_rs) with
  |  $\emptyset \Rightarrow \emptyset$ 
  |  $[rs] \Rightarrow \text{match alloc\_frame}(sz, pos, rs, m) \text{ with}$ 
    |  $\emptyset \Rightarrow \emptyset$ 
    |  $[(rs', m')] \Rightarrow [(rs' \{ r_{14} \leftarrow \text{abstract}(pc) + 1, pc \leftarrow rs'[r_0] \}, m')]$ 

```


We then define a boolean predicate *is_final_state* to describe a well-formed final state of the `jited` code.

$$\begin{aligned} \text{is_final_state}(rs : SReg) : \text{bool} = & \quad rs[pc] == \text{abstract}(r_{14}) \quad \&\& \\ & \quad rs[sp] == \text{abstract}(sp) \quad \&\& \\ & \quad (\forall i. 4 \leq i \leq 11 \rightarrow rs[r_i] == \text{abstract}(r_i)) \end{aligned}$$

The predicate *is_final_state* stipulates that, before switching to the rBPF-32 level for interpreting other non-alu rBPF-32 instruction, the final state should satisfy:

- *pc*: The program counter should hold the return address stored in r_{14} .
- *sp*: The newly allocated stack frame should be free.
- *calling convention*: All callee-save registers should have their initial values.

6.3.2 Transition Semantics of rBPF-32-JIT

rBPF-32-JIT  refines the aforementioned transition system by the following aspects:

- *State*: The program state is represented as a quadruple $rbpf32_arm_state ::= (b, SR, R, M)$ where the boolean flag b indicates the transition system is at the rBPF-32 level ($b = false$) or at the ARM level ($b = true$), SR is a symbolic ARM register map, R is a rBPF-32 register map, and M denotes the CompCert memory.
- *Additional CompCert Blocks*: As mentioned in [Section 6.2.1](#), there are two special blocks in the CompCert memory M : *jit_blk* used for storing the `jited` code and *st_blk* for representing a global state, including the rBPF-32 register map R . In particular, *st_blk* provides a low-level abstraction of the rBPF-32 register map: The rBPF-32 interpreter operates on the rBPF-32 register map separately from the CompCert memory, making rBPF-32 register operations more abstract and corresponding proofs easier. While the ARM interpreter requires the rBPF-32 register map to be stored in a CompCert memory block.

To synchronize *st_blk* and the rBPF-32 register map, the Gallina function `copy_to` copies all rBPF-32 registers' values into *st_blk* before the switch from the rBPF-32 interpreter to the ARM interpreter, and `copy_from` performs the opposite operation, updating rBPF-32 registers according to the new content in *st_blk* after the switch from the ARM interpreter to the rBPF-32 interpreter.

$$\begin{aligned}
\text{copy_to}(R, st_blk, m) = & \\
& \mathbf{match\ store}(Mint32, m, st_blk, 4, R[R_0]) \mathbf{with} \\
& | \emptyset \Rightarrow \emptyset \\
& | [m_1] \Rightarrow \mathbf{match\ store}(Mint32, m_1, st_blk, 8, R[R_1]) \mathbf{with} \\
& \quad \dots \\
& | [m_{10}] \Rightarrow \mathbf{store}(Mint32, m_{10}, st_blk, 48, R[PC])
\end{aligned}$$

$$\begin{aligned}
\text{copy_from}(R, st_blk, m) = & \\
& \mathbf{match\ load}(Mint32, m, st_blk, 4) \mathbf{with} \\
& | \emptyset \Rightarrow \emptyset \\
& | [v_0] \Rightarrow \mathbf{match\ load}(Mint32, m, st_blk, 8) \mathbf{with} \\
& \quad \dots \\
& | [v_{10}] \Rightarrow \mathbf{match\ load}(Mint32, m_{10}, st_blk, 48) \mathbf{with} \\
& \quad | \emptyset \Rightarrow \emptyset \\
& \quad | [v_{pc}] \Rightarrow [R\{R_0 \leftarrow v_0, \dots, R_{10} \leftarrow v_{10}, PC \leftarrow v_{pc}\}]
\end{aligned}$$

- *Transition Relation*: The new transition relation, denoted as $G, C, MR, KV, st_blk, jit_blk \vdash st \xrightarrow{t} st'$, introduces additional global information KV , which is a *key_value* list used to link a sequence of **Alu32** binary and its generated **jited** binary stored in jit_blk , as explained in [Section 6.4.1](#).
- *Transition Rules*: The transition rules for **non-Alu** cases are directly inherited from the original rBPF-32 transition system where the flags in both states are set to *false*, and the symbolic ARM register map is unchanged.

$$\begin{aligned}
& R[PC] = \mathbf{Vint}(n) \quad C[n] = [ins] \quad ins \in \{\mathbf{Ja32\ ofs}, \mathbf{Jump32\ cmp\ dst\ src\ ofs}, \\
& \quad \mathbf{Neg32\ dst}, \mathbf{Load32\ ck\ dst\ reg\ ofs}, \mathbf{Store32\ ck\ dst\ src\ ofs}, \mathbf{Call32\ imm}\} \\
(\text{Non_alu}) & \frac{\dots}{G, C, MR, KV, st_blk, jit_blk \vdash (false, SR, R, M) \xrightarrow{t} (false, SR, R', M')}
\end{aligned}$$

Three rules related to our JIT compiler are introduced to replace the old *alu* rule.

The first rule *JIT_init* finds an *alu* instruction that the corresponding element in *KV* records an offset. It updates the rBPF-32 register map to the block *st_blk* (*copy_to*), switches the transition from rBPF-32 level to ARM level, and initializes an ARM state using the *init_state* function. The *signature* consists of argument types, a return type, and the default calling convention. The arguments list $\overrightarrow{args}(ofs) = [\mathbf{Vptr}(jit_blk, ofs), \mathbf{Vptr}(st_blk, 0)]$ records the values of the corresponding arguments where the first argument, relying on the offset *ofs*, is the start address of *jited* code, and the second one points to the start address of a global state.

$$\begin{array}{c}
 R[PC] = \mathbf{Vint}(n) \quad C[n] = [\mathbf{alu} \ op \ dst \ src] \quad KV[n] = [ofs] \\
 \text{copy_to}(R, st_blk, M) = [M'] \\
 \text{init_state}(sig, \overrightarrow{args}(ofs), sz, pos, M') = [(SR', M'')] \\
 (JIT_init) \frac{}{G, C, MR, KV, st_blk, jit_blk \vdash (false, SR, R, M) \xrightarrow{\epsilon} (true, SR', R', M'')}
 \end{array}$$

The second transition rule *JIT_final* performs the opposite switch: from ARM level to rBPF-32 level, when the symbolic register map *SR* represents the final state. It updates the rBPF-32 register map with the new content of the block *st_blk* in memory *M*. The last

$$\begin{array}{c}
 R[PC] = \mathbf{Vint}(n) \quad C[n] = [\mathbf{alu} \ op \ dst \ src] \quad \mathbf{is_final_state}(SR) = true \\
 \text{copy_from}(R, st_blk, M) = [R'] \\
 (JIT_final) \frac{}{G, C, MR, KV, st_blk, jit_blk \vdash (true, SR, R, M) \xrightarrow{\epsilon} (false, SR, R', M)}
 \end{array}$$

rule *JIT* handles the normal transition of one CompCert ARM instruction in a symbolic way, *i.e.*, calling *symbolic_transf*.

$$\begin{array}{c}
 R[PC] = \mathbf{Vint}(n) \quad C[n] = [\mathbf{alu} \ op \ dst \ src] \quad \mathbf{find_instr}(SR[pc], M) = [ins] \\
 \text{symbolic_transf}(ins, SR, M) = [(SR', M')] \\
 (JIT) \frac{}{G, C, MR, KV, st_blk, jit_blk \vdash (true, SR, R, M) \xrightarrow{\epsilon} (true, SR', R', M')}
 \end{array}$$

6.4 Hybrid JIT Interpreter

This section introduces a hybridly accelerated virtual machine HAVM \clubsuit , *i.e.*, an interpreter of refined rBPF-32, and demonstrates how to extract executable C implementation of the JIT_{ALU} compiler and the hybrid interpreter. Section 6.4.1 provides a global view of the entire hybrid JIT interpreter. Section 6.4.2 presents our CompCert ARM interpreter variant. Section 6.4.3 defines a refined JIT_{ALU} compiler with a monadic state. Section 6.4.4 implements a hybrid rBPF-32 interpreter in monadic form, capable of interpreting the refined rBPF-32 (including `jited` code). Finally, Section 6.4.5 leverages the workflow discussed in Chapter 4 to generate C code for both the JIT_{ALU} compiler and the corresponding interpreter.

6.4.1 Overview

HAVM follows a hybrid design where:

- *Arithmetic instructions* (rBPF-32 `Alu32`) are directly translated into the target ARM binary and executed natively for achieving better performance.
- *Non-Arithmetic instructions* (*e.g.*, `branch`, `memory`, etc) are simply executed by the host rBPF-32 virtual machine to balance hypothetical performance gains with the cost of necessary defensive code.

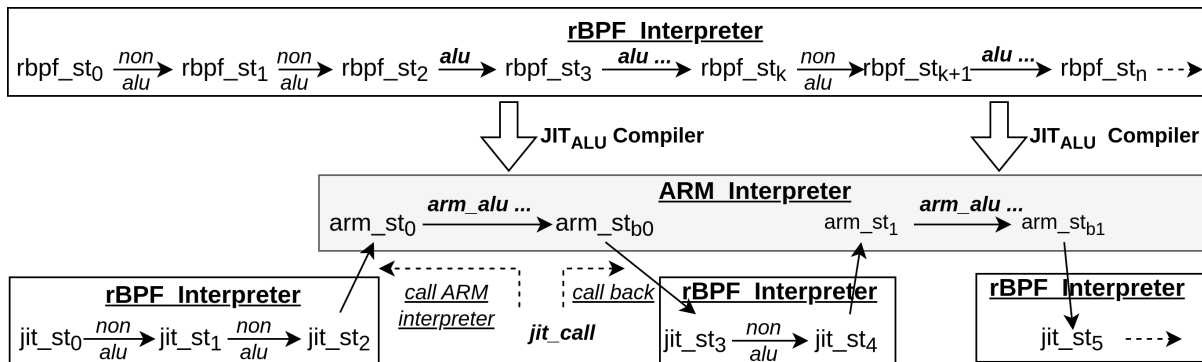


Figure 6.5 – HAVM Overview

Figure 6.5 depicts the program transformation performed by HAVM, starting from a given rBPF script (top). The JIT_{ALU} compiler scans linearly the rBPF input script to find sequences of arithmetic `alu` operations. For each such sequence, it generates an

equivalent sequence of `ARM_alu` operations, whose formal semantics is adapted from the existing CompCert ARM backend.

The last step (bottom) consists of interleaving the interpretation of non-`alu` operations, in the VM, with the execution of the `jited ARM_alu` binary sequences. This interleaving is performed in two steps by a glue function named `"jit_call"`. The first step consists of jumping from state `jit_st2` in the `rBPF_interpreter` to the translated state `arm_st0` in the `jited` binary. The second step is to callback the interpreter in state `jit_st3` from the ARM state `arm_stb0`.

The complete `JITALU` compiler for HAVM, as depicted in [Figure 6.5](#), operates in two steps: 1) Selecting a sequence of consecutive `Alu32` instructions. 2) Translating the `rBPF-32 Alu32` instructions into ARM binary code.

The selection in the first step follows the principles:

- *entry point*: the first `Alu32` instruction is special, and as shown in [Figure 6.6](#), it is one of the three cases: 1) its index is 0, 2) its previous instruction is non-`Alu32`, or 3) there exists a jump instruction branching to this `Alu32` that its previous instruction can be `Alu32` or non-`Alu32`. The index of such a `Alu32` instruction is named *entry point*.
- *tail*: the next instruction of the tail of the selected sequence is non-`Alu32`.

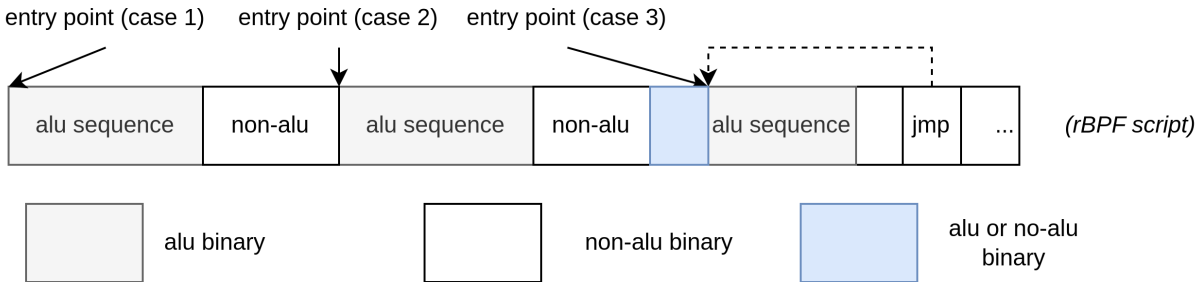


Figure 6.6 – Three cases of *entry point*

The first step provides a global mapping relation from each sequence of `Alu32` instructions to the corresponding `jited` binary code. This mapping is stored in a key-value list denoted as `KV`. Given a sequence of `Alu32` instructions `l` with an entry point `n` in the input `rBPF` script, the key in the list `KV` is `n`, and the corresponding value records the start address of the `jited` code generated from `l`. The second step has been introduced in [Section 6.2](#).

6.4.2 CompCert ARM Interpreter

To ensure correct interpretation of the `jited` code, we have designed a symbolic variant of the CompCert ARM interpreter. This variant leverages the existing CompCert ARM transition functions to execute user-specific ARM binary code.

The function `bin_exec` is used to implement this symbolic CompCert ARM module. It operates as follows:

```

bin_exec(fuel, sig, args, sz, pos, m) =
  match init_state(sig, args, sz, pos, m) with
  |  $\emptyset \Rightarrow \emptyset$ 
  |  $[(rs', m')]$   $\Rightarrow$  bin_interp(fuel, rs', m')

```

Its parameters include:

- `fuel`, which ensures the termination of its recursive sub-function `bin_interp`.
- `sig`, representing the signature of the arguments used by the input ARM binary code.
- `args`, the corresponding argument list. We stipulate that the maximum number of arguments is 5, where `init_state` responds to copy those values into $r_0 - r_4$, respectively.
- `sz`, indicating the size of the allocated stack frame.
- `pos`, specifying where the old stack pointer should be stored in the new stack frame.
- `m`, the CompCert memory.

First, `bin_exec` uses `init_state`, as described in [Section 6.3.1](#), to create a proper ARM environment that includes a fixed-size stack frame and a new state for the ARM register map. It then calls `bin_interp` recursively to interpret ARM binary code until it achieves a final state (*i.e.*, `is_final_state(rs) = true`) and returns r_0 's value, or until the fuel is exhausted. In each iteration, `find_instr` locates the instruction pointed to by the program counter `pc` and decodes it using the decode function discussed in [Section 6.3.1](#). If the binary instruction decodes successfully, `bin_interp` then calls a symbolic ARM transition function `symbolic_transf` to execute the instruction and proceeds to the next

instruction if no errors occur.

```

bin_interp(fuel, rs, m) =
  if is_final_state(rs) then [(rs[r0], m)]
  else if fuel == 0 then ∅
  else match find_instr(rs[pc], m) with
    | ∅ ⇒ ∅
    | [ins] ⇒ match symbolic_transf(ins, rs, m) with
      | ∅ ⇒ ∅
      | [(rs', m')] ⇒ bin_interp(fuel - 1, rs', m')

```

6.4.3 Monadic JIT Compiler

Firstly, a JIT state is a 9-tuple $jit_state ::= (C, C_len, M, Ld_list, St_list, jit_list, jit_ofs, jit_len, KV)$ where:

- C : a list of rBPF binary code.
- C_len : the fixed length of C .
- M : CompCert memory model.
- Ld_list : a list of load register with size 11.
- St_list : a list of store register with size 11.
- jit_list : a list used to store the `jited` code.
- jit_ofs : an offset pointing to the next unused space in the list jit_list .
- jit_len : the pre-allocated size of jit_list .
- KV : a `key_value` list with size C_len , where its `index` is a key representing an entry point (see [Figure 6.6](#)) and the corresponding value $KV[index]$ points an offset in jit_list .

Ld_list and St_list are used to store some registers (index) at the Load and Store Stages of JIT_{ALU} . All `jited` binary instructions are stored in the list jit_list with a fixed size of jit_len (corresponding to a C array), and jit_ofs counts the number of binary instructions generated by JIT_{ALU} .

The monadic JIT compiler $jit_alu : M\ bool\ \clubsuit$, based on the design in [Section 6.2](#), accepts an initial state, it either crashes or returns a new state and a boolean result. A


result of *false* indicates that the compiler failed to complete the JIT process for some reason, *e.g.*, *jit_list* being full ($jit_ofs \geq jit_len$).

6.4.4 Hybrid rBPF-32 Interpreter

This section introduces how to interpret the composition of an rBPF binary script and the generated `jited` ARM binary code in a monadic way.

First, the monadic state, a 12-tuple $hybrid_state ::= (C, C_len, PC, BR, f, M, MRs, mrs_num, KV, jit_ptr, jit_len, st_ptr)$, refines CertrBPF's state *rbpf_state* (see [Section 5.1.2](#)) in the following aspects:

- Considering that our CompCert ARM interpreter uses CompCert memory functions to read or write `jited` code and the rBPF-32 register map, this new CompCert Memory *M* introduces two special blocks: *jit_blk* and *st_blk* where $jit_ptr = Vptr(jit_blk, 0)$ and $st_ptr = Vptr(st_blk, 0)$.
- the new state also adds three JIT related fields derived from the JIT state. They are the *key_value* list *KV*, the pointer *jit_ptr* pointing to the beginning of `jited` binary code, and the fixed size *jit_len*.

Our hybrid rBPF-32 interpreter `hybrid_interp` , based on the new monadic state, inherits most of its structure, *e.g.*, the *interp* function and the *Jump*-related implementation in the *step* function, directly from [Section 5.1.3](#). The only difference is the call to a glue function `jit_call` to interpret `jited` code when *hybrid_step* finds an rBPF *alu* instruction.

```

hybrid_step(hst) =
  match hst.C[st.PC] with
  | Alu op dst src  $\Rightarrow$  jit_call(hst.KV[hst.PC], hst)
  | Jump cmp dst src ofs  $\Rightarrow$  ../..
  | ../..

```

`jit_call` is formally defined as an instantiate of our symbolic CompCert ARM module `bin_exec`, where:

- *fuel* is set to the size of *jit_blk*.

- *signature* and the argument list $\overrightarrow{args}(ofs)$ are the same as the ones used in the *JIT* transition rule (Section 6.3.2).
- The newly allocated stack frame has a fixed size of 48, where the old *sp* is stored at position 0, as shown in Figure 6.3.
- The final state updates the CompCert memory with the returned value of `bin_exec`.

```

jit_call(ofs, hst) =
  match bin_exec(hst.jit_len, sig,  $\overrightarrow{args}(ofs)$ , 48, 0, hst.M) with
  |  $\emptyset \Rightarrow \emptyset$ 
  |  $[(\_, N)] \Rightarrow hst\{M \leftarrow N\}$ 

```

6.4.5 rBPF-32 C Implementation

Following the aforementioned workflow Chapter 4, we construct synthesis models of our `JITALU` Compiler and the hybrid rBPF-32 interpreter. We use ∂x to extract C implementation. There are two ways to implement the `jit_call` function: either by adding a CompCert build-in function, or considering `jit_call` as an external C function. Here we mainly discuss the second approach.

jit_call Implementation. The external C function `jit_call` sets up a properly initialized ARM calling environment, where:

- r_0 : Points to the correct location of the `jited` ARM binary.
- r_1 : Contains the start address of the JIT state.
- *sp*: Indicates the start address of the newly allocated stack frame.
- r_{14} : Holds the return address from `jit_call`.
- *pc*: Equals r_0 for executing the first given `jited` ARM binary.

`jit_call` accepts two arguments: *ofs* for the offset of the corresponding `jited` code in a pre-allocated array *jit_ptr*, and *st* for the global state of the hybrid interpreter. `jit_call` first switches the processor mode into THUMB mode, *i.e.*, (line 4) `'orr rd #0x1'` where the register r_d points to the given address of the `jited` code. Then it allocates a new stack frame (line 5- line 7) where location `[sp, # 0]` is used to store the old stack pointer. Consequently, `jit_call` uses a `move` instruction (line 8) to branch execution to

the offset of the `jited` block because `mov` doesn't modify the value of the linking register, allowing it to correctly jump to the return address.

Finally, line 9 declares that return value is stored into a temporary variable `res`. The ARM register `r1` points to a pointer to the global state, and line 10 indicates `r0` has the address pointing to the `jited` code. `jit_call` doesn't return anything because the `jited` code has updated the global state.

```

1 void jit_call(unsigned int ofs, struct hybrid_state* st){
2     int res;
3     asm volatile (
4         "orr %[input_0], #0x1\n\t" // set THUMB mode
5         "mov r12, sp\n\t"         // save old sp
6         "sub sp, sp, #48\n\t"     // allocate a new frame
7         "str r12, [sp, #0]\n\t"   // save old sp
8         "mov pc, %[input_0]\n\t" // mov: change pc and preserve lr
9     : [result] "=r" (res) : [input_1] "r" (st),
10      [input_0] "r" (jit_ptr + ofs) : "cc");
11     return ;
12 }

```

HAVM: a Hybrid Interpreter. HAVM includes the `JITALU` compiler and a hybrid interpreter. For the `alu` case, it contains two functions: the first one calculates an offset (`ofs`) of the `jited` ARM block according to the `rBPF PC`, and `jit_call`. In practice, the `jit_blk` is defined by a C array and is hence declared in the `JIT` state because it is an effectful data structure.

```

void step(struct hybrid_state* st){ /* ... */
    switch (opcode) {
        case ALU32:
            /*compute ofs */
            jit_call(ofs, st);
        case Jump32: ...
        ...:
    }
}

void hybrid_interpreter(struct hybrid_state* st, unsigned int fuel){
    /* ... if (fuel == 0) { ... } */
    step(st);
    /* ... hybrid_interpreter(st, fuel -1); */
}

```

6.5 Discussion

The previous sections introduce the design of our JIT_{ALU} compiler and the hybrid rBPF-32 interpreter, the remained proof details are discussed in this section and the Coq mechanized validation is a work-in-process.

6.5.1 Proof Overview

As shown in [Figure 6.7](#), the proofs, represented by \sim for simulation relations and \models for critical properties, include a collection of proofs for JIT-related models and a set of proofs for rBPF-32-related models.

For the JIT compiler, the proofs consist of:

- Establishing equivalence between the JIT_{ALU} compiler (introduced in [Section 6.2](#)) and its monadic model (mentioned in [Section 6.4.3](#)),
- Ensuring that the monadic JIT_{ALU} (proof model) is fault-isolated, *i.e.*, no undefined behaviours.
- Proving the simulation relation among its proof, synthesis, and clight models.

For the rBPF-32 interpreter, the proofs involve:

- Proving the simulation relation between two transition systems: the original rBPF-32 semantics (defined in [Section 6.1](#)) and that of the refined rBPF-32-JIT (defined in [Section 6.3](#)). This simulation proof is exactly the JIT_{ALU} correctness.
- Establishing the equivalence between the refined semantics and the corresponding interpreter (proof model).
- Following the verification workflow to guarantee that the rBPF-32-JIT proof model is isolated, and all related models are proven equivalent.

Our discussion primarily centers on the JIT correctness theorem since the rest can be easily mechanized checked by reusing the previously mentioned proof strategies (see [Chapter 5](#)).

We first define the simulation relation $\sim \subseteq \text{rbpf32_state} \times \text{rbpf32_arm_state}$. For simplification, we mainly discuss the relation between register maps, where the source state has only one rBPF-32-JIT register map and the target state includes a pair of register maps: rBPF-32-JIT and ARM. As depicted in [Figure 6.8](#), \sim contains two cases that depend on the current rBPF-32-JIT instruction (*i.e.*, $\lfloor \text{st.C}[\text{st.R}[PC]] \rfloor$).

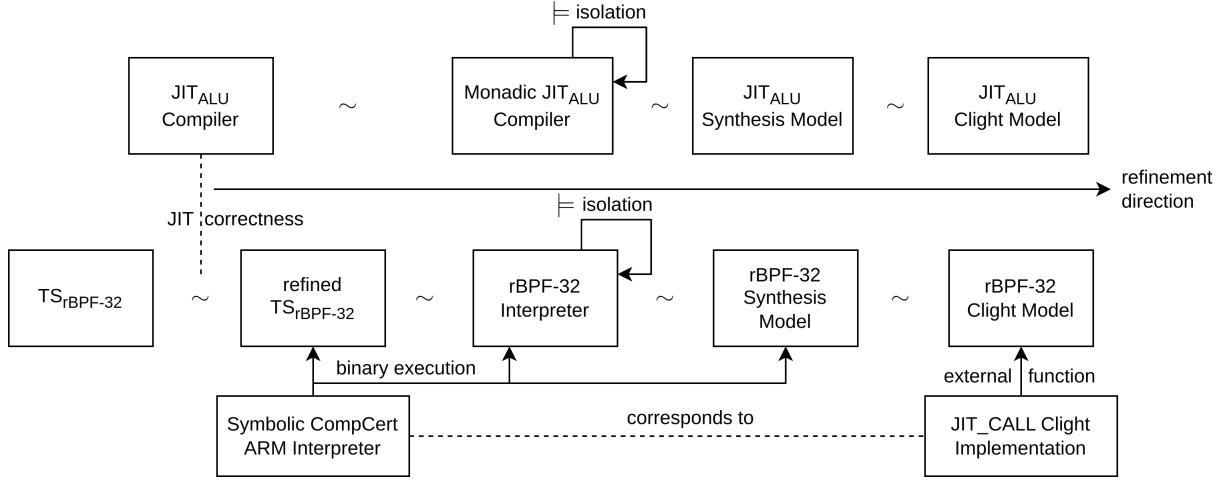


Figure 6.7 – JIT Proof Overview

- *non-Alu32*: If the current instruction is an rBPF-32-JIT *non-Alu32* instruction, the forward simulation produces identical transition behaviours in source and target systems. In this case, two rBPF-32-JIT register maps are updated consistently, and the ARM register map in the target state is omitted, *i.e.*, \sim_{rbpf} .

$$\sim_{rbpf}(st_0, st_1) \stackrel{\text{def}}{=} \forall r. st_0.R[r] = st_1.R[r]$$

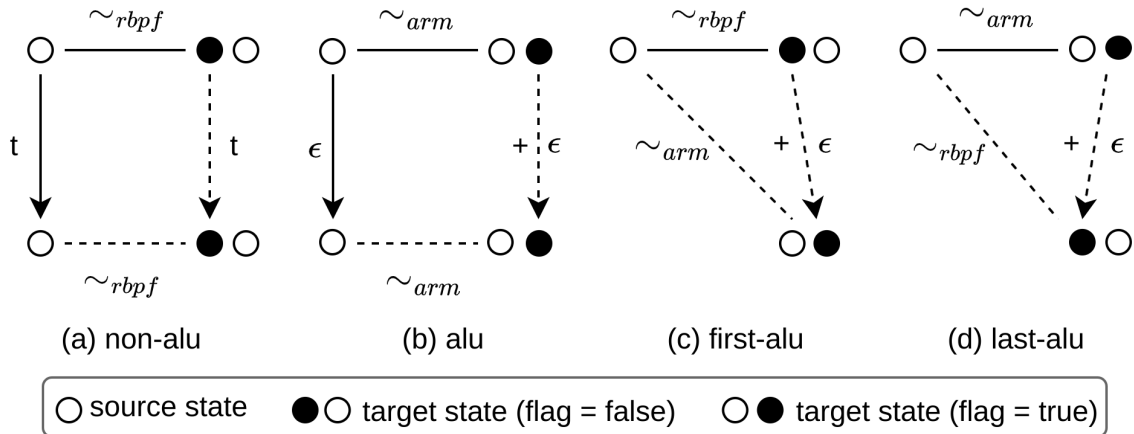


Figure 6.8 – JIT Simulation Diagrams

- *Alu32*: For each rBPF *Alu32* instruction, the forward simulation follows a standard CompCert forward approach, where the source undergoes one transition of

rBPF-32-JIT semantics, and the target undergoes multiple transition steps using the symbolic CompCert ARM transition functions. The simulation relation \sim_{arm} is defined as follows: For a list of synchronized rBPF-32-JIT registers L with no duplicated elements, each register in L always has the same value as the corresponding ARM register. For registers not in L , they are identical between source and target register maps R . We rewrite *ireg_of_reg* for computing the corresponding ARM registers, *e.g.*, ‘*ireg_of_reg*(R_0) = r_0 ’.

$$\begin{aligned} \sim_{\text{arm}}(st_0, st_1, LR) &\stackrel{\text{def}}{=} \forall r. \\ &(\text{List.In } r \text{ } LR \rightarrow st_0.R[r] = st_1.SR[\textit{ireg_of_reg}(r)]) \wedge \\ &(\neg \text{List.In } r \text{ } LR \rightarrow st_0.R[r] = st_1.R[r]) \end{aligned}$$

- *First Alu*: If the current instruction is the first *Alu* instruction of a sequence of *Alu* instructions, *i.e.*, *entry point*, the forward simulation introduces an additional triangular form to switch from the rBPF level to the ARM level. The process, corresponding to the Pre, Spilling, and Load Stages of JIT_{ALU} , constructs the synchronous registers LR from an empty set according to the Load Stage.
- *Last Alu*: If the current instruction is the last *Alu* instruction of a sequence of *Alu* instructions, the forward simulation employs another triangular form to switch from the ARM level to the rBPF level. The process, corresponding to the Store, Reloading, and Post Stages of JIT_{ALU} , frees the synchronous registers LR according to the Store Stage.

6.5.2 Defensive JIT_{ALU}

In the previous section, our JIT_{ALU} compiler skips four special rBPF instructions, as seen in [Table 6.1](#), because they require runtime checks, *e.g.*, division-by-zero of the source register in an rBPF *DIV* register-related instruction. In this section, we discuss how to extend JIT_{ALU} to generate defensive `jited` code.

The main challenge of this defensive JIT_{ALU} is to interrupt the ARM interpreter correctly and return a proper rBPF flag when `jited` code captures runtime errors.

Firstly, we re-organize the structure of the JIT_{ALU} procedure. Reset and Post Stages are moved to the beginning. The change provides a fixed address for exception cases. When `jited` code finds an exception, the corresponding rBPF error flag is updated to

the flag field of a global state (*e.g.*, the first four bytes in [Figure 6.4](#)). Subsequently, the ARM interpreter jumps to the start address of the Reset Stage to ensure ARM calling convention. An additional Jump Stage is introduced to ensure `jited` code also branches to the Reset Stage in normal cases.

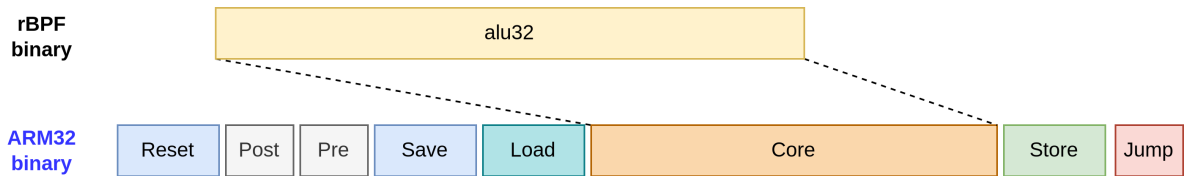


Figure 6.9 – JIT defensive procedure

Then, we extend the core mapping rules to handle three shift instructions and the rBPF `DIV` register-related instruction.

As demonstrated in [Figure 6.10](#), the three rBPF-32 shift instructions follow the same pattern to generate additional defensive ARM code for runtime checks. In the figure, various colors distinguish instructions with distinct purposes: rBPF-32 instructions are highlighted in orange, ARM comparison instructions in grey, ARM branch instructions in violet, while instructions marked in red are employed to interrupt the execution of `JITALU`. The green instructions represent ARM instructions translated from rBPF-32 instructions. Each solid arrow represents a transition from an ARM branch instruction to another ARM instruction. These arrows, along with labels denoting boolean conditions, signify that the destination instruction may be executed after the source branch instruction, depending on the outcome of the preceding comparison instruction. In addition, the box sizes in [Figure 6.10](#) represent the different instruction widths in the target machine, such as rBPF-32 instructions are 64-bit, ARM comparison instructions are 16-bit, and the remaining ARM instructions are 32-bit.

Our shift mapping rule initially compares the source register r_s with the constant 32. If the value of r_s is (signed) less than 32, *i.e.*, ' $r_s < 32$ ', pc is set to the subsequent ARM comparison instruction '`cmp rs #0`', else (' $r_s \geq 32$ '), pc goes to the next instruction to interrupt the execution of `JITALU`, *i.e.*, the red color instructions in [Figure 6.10](#). The following comparison instruction `cmp rs #0` and the branch instruction `blt #(-9)` also adjust pc to the instructions colored in red when ' $r_s < 0$ '. Conversely, (' $0 \leq r_s < 32$ ') pc branches to the corresponding ARM shift instructions responsible for normal shift operations. The instruction '`movw r11, #11`' copies the encoding value of `BPF_ILLEGAL_SHIFT` into r_{11} , and '`str r11, [r12, #0]`' updates the BPF flag (located at the start address of the

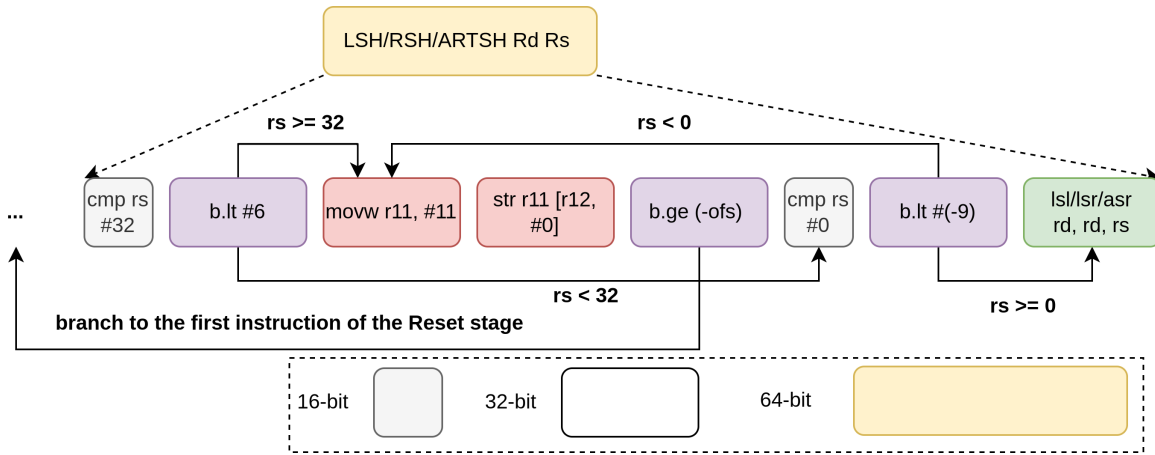


Figure 6.10 – JIT defensive procedure: shift instructions

global state, *i.e.*, offset 0) with the value of r_{11} .

Similar to the rBPF-32-JIT shift instructions, the ‘*DIV*’ instruction is also translated to a sequence of defensive ARM code, as shown in Figure 6.11.

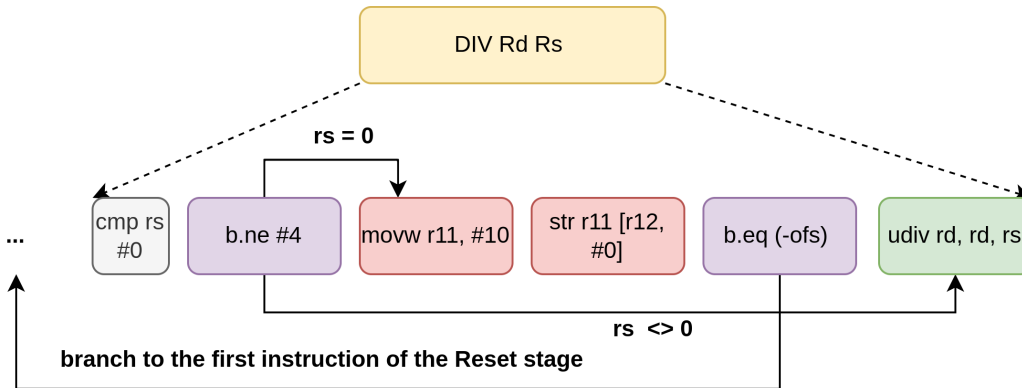


Figure 6.11 – JIT defensive procedure: *DIV* instruction

To calculate the branch offset, consider the example ‘*b.lt* 6,’ which executes a branch based on the condition *lt*. The offset is computed using the following formula²:

$$offset = (X - 4) / 2$$

Where X represents the number of bytes to jump. *e.g.*, if the *cmp* instruction is encoded in 16 bits, while *b*, *movw*, and *str* instructions are encoded in 32 bits, then

2. The real-world THUMB interpreter adopts $PC \leftarrow PC + (offset * 2) + 4$ for *b.c* offset

$X = b.lt(4) + movw(4) + str(4) + b.lt(4)$, resulting in a total of 16. Consequently, $offset = (16 - 4)/2$, which equals 6. Thus, We get 'b.lt 6' in [Figure 6.10](#).

EVALUATION

In this chapter, we integrate CertrBPF as a drop-in replacement for the existing non-verified module optimized for size (vanilla-rBPF) in the IoT operating system RIOT to provide the expected femto-container functionalities [ZB20]. We then evaluate CertrBPF performance compared to vanilla-rBPF, compiled and running on real IoT hardware.

7.1 Implementation

7.1.1 Coq Implementation

We use the tool `cloc` (full name, Count Lines of Code) developed by Al Danial to count our code size, and the results are presented in [Table 7.1](#):

Table 7.1 – Coq code statistics of CertrBPF

Module	Interpreter (loc)	Verifier (loc)	JIT (loc)
Proof Model	2445	1459	2074
Properties	4885	547	9617*
Synthesis Model	3285		
Equivalence	635		
Simulation	10820	8331	
(Total)	22070	10337	

* proof underway.

- *CertrBPF Interpreter*: The proof model of the interpreter ([Section 5.1](#)) consists of 2.4k lines of Coq code and the corresponding isolation proof ([Section 5.1.4](#)) is more than 4.8k lines long. The synthesis model, [Section 5.2](#), is approx. 3.2k lines long and the equivalence theorem is completed by 0.6k proof code. The final step ([Section 5.3](#)) includes 10.8k translation validation proofs between the Gallina specification and the extracted Clight model.

- *CertrBPF Verifier*: As for the CertrBPF verifier (Section 5.4), the proof and synthesis models sport 1.4k lines of Coq code. The corresponding proofs are more than 0.5k long and the last simulation proof is about 8.3k long.
- *CertrBPF JIT*: The JIT compiler implementation consists of around 2k lines, and the corresponding proof (currently underway) is about 9k lines.

In addition, the *Clightlogic* implementation has 4.4k lines of Coq code.

7.1.2 C Implementation

The final C implementation is automatically extracted from our Gallina specification using ∂x . Table 7.2 provides a static comparison between the CertrBPF C implementation and vanilla rBPF.

Table 7.2 – C code statistics of CertrBPF

	Vanilla-rBPF	CertrBPF
Interpreter	730	854
Verifier	68	548
JIT Compiler		1336

We first notice that the extracted model consistently results in more lines of C code. The main reason is that CertrBPF adopts a monadic-style code design, while vanilla rBPF employs a common programming style. For each atomic state-related operation, such as register updates, CertrBPF models it as a monadic function in Gallina and extracts a monadic C function, whereas vanilla rBPF uses equivalent C operators.

```

/**r BPF_ADD_REG example: */
/* vanilla rBPF: uint64_t *src, uint64_t *dst */
    *dst += *src;
    ...
/* CertrBPF: unsigned long long dst64, unsigned long long src64, unsigned int dst */
    upd_reg(st, dst, dst64 + src64);
    return ;
    ...

```

Next, we observe that the generated C code of CertrBPF includes many small functions that only perform very simple tasks, *e.g.*, *eval_reg* (reading a register value from register

map) and `upd_pc` (writing a new value to rBPF program counter). Calls to those functions can produce extra overhead. To address this, we have declared the `always_inline` attribute for these simple functions to improve performance.

```
static __attribute__((always_inline)) inline void upd_pc(struct bpf_state* st,
→ unsigned int pc) {
    ...
}
```

7.2 Experiment

Our experiments involve the original non-verified rBPF interpreter (*i.e.*, vanilla-rBPF) and the automatically extracted CertrBPF interpreter (without RIOT’s API) as well as the HAVM implementation (JIT compiler + hybrid interpreter). Our measurements focus on the memory requirements (*RQ1*) of these virtual machines and the instruction execution throughput (*RQ2-RQ4*).

7.2.1 Experimental Evaluation Setup

Our experiments are conducted using the *nrf52840dk* support board, which uses an Arm Cortex-M4 microcontroller, a popular 32-bit architecture (arm-v7m). The code is compiled using the Arm GNU toolchain version 12.2 with level 2 optimization enabled. We utilize the following GCC compiler options:

- `-foptimize-sibling-calls`: Optimizes all tail-recursive calls to bound the stack usage.
- `-falign-functions=16`: Reduces performance variation caused by the instruction cache on the device.
- `-fwrapv`, `-fwrapv-pointer`: Enables both signed and pointer arithmetic wraps according to the two’s-complement encoding.
- `-fno-strict-aliasing`: Disables aliasing assumption.

The `-foptimize-sibling-calls` option is particularly critical for our isolation theorem, as it relies on the implicit CompCert assumption that the stack cannot overflow. The last three options are passed for the purpose of avoiding a possible mismatch between the CompCert semantics and the GCC semantics.

7.2.2 Benchmarks

Our evaluation includes two types of benchmarks:

- *Micro-benchmarks*: single instructions from the arithmetic logic unit (ALU), for memory access (MEM) and branch instructions, with a mix of register and immediate value for the operands. The micro-benchmarks are used to measure the performance of core instructions. For the precise measurement, each micro-benchmark consists of 1000 single identical instruction calls with a single return statement to make the application exit.
- *Macro-benchmarks*: actual benchmarks for different purposes: i) pure computation tasks, such as *incr*, *square* and *fib*; ii) memory read/write tasks, for instance *sock_buf* and *memcpy(n)*; iii) actual benchmarks from rBPF: *fletcher32* algorithm, etc. The source code for *macro-benchmarks* is written in C and compiled to BPF binary with proper compiler options, e.g., LLVM-eBPF's '-mattr=+alu32' for generating instructions that operate on 32-bit subregisters.

7.2.3 Research Question: Memory Footprint

RQ1: Do CertrBPF and HAVM meet the memory requirements of IoT hardware compared to vanilla-rBPF?

We first evaluate the memory footprint of the CertrBPF interpreter and the HAVM implementation in comparison to vanilla-rBPF. We measure i) *Flash size*, which includes all read-only data, including the actual code; ii) *Stack*, representing the approximate ram used for stack space.

To directly measure the memory footprint, we examine the executable binary file *.elf* generated from a RIOT-OS application that includes the rBPF virtual machine module written in C. ELF files are selected because they are clean: the GCC compiler performs optimization, and the linker *ld* removes all dead functions.

We use the tool *arm-none-eabi-nm* to measure flash size with the options '-S' for printing size, '-line-numbers' for printing location in the binary, and '-radix=d' for values in decimal format. The *grep* command collects all compiled functions of the virtual machine *Y* module of the RIOT-OS application *X*, and *awk* sums the sizes.

```
$ arm-none-eabi-nm X.elf -S --line-numbers --radix=d |grep \\\Y\\ | awk '{ SUM += $2}  
↪ END{ print SUM}'
```

We use the GCC option `-fstack-usage` to calculate worst-case stack usage of individual functions, and the results are recorded in the generated `.su` files.

Size	Vanilla-rBPF	CertrBPF	CertrBPF (opt)	HAVM	
				Interpreter	JIT
Flash	2018 B	1502 B	2114 B	702 B	8756 B
Stack	356 B	68 B	96 B	68 B	164 B

Table 7.3 – Memory footprint of rBPF engines

We present a comparison of memory requirements for different implementations in [Table 7.3](#).

Key take-away. In terms of Flash,

- CertrBPF actually reduces the footprint by a 25% decrease on Cortex-M, compared to Vanilla-rBPF. One reason is that calls to the RIOT API are currently not supported by CertrBPF;
- The `check_mem` optimized CertrBPF increases the footprint due to the additional cache field;
- HAVM requires significantly more Flash compared to other interpreters. This is primarily because that both a regular eBPF interpreter and additional JIT compilation code are included. The HAVM interpreter has the smallest Flash because some of its sub-functions are shared with the JIT compiler and they are counted in the latter.

In terms of Stack, CertrBPF has the lowest stack usage. Vanilla-rBPF has an extra call module that occupies approximately 108 B. The optimized CertrBPF increases stack usage due to the cache field. Last, HAVM consists of an interpreter and a JIT compiler, which imposes additional stack requirements on top of what is shown in [Table 7.3](#).

In conclusion, all implementations require a comparable amount of Flash and Stack for the rBPF virtual machine.

7.2.4 Research Question: CertrBPF Interpreter Performance

RQ2: Can the verified CertrBPF C implementation outperform vanilla-rBPF?

First, we evaluate the performance of micro-benchmarks for core operations, as shown in [Figure 7.1](#), which includes five ALU64 instructions, three memory instructions, and

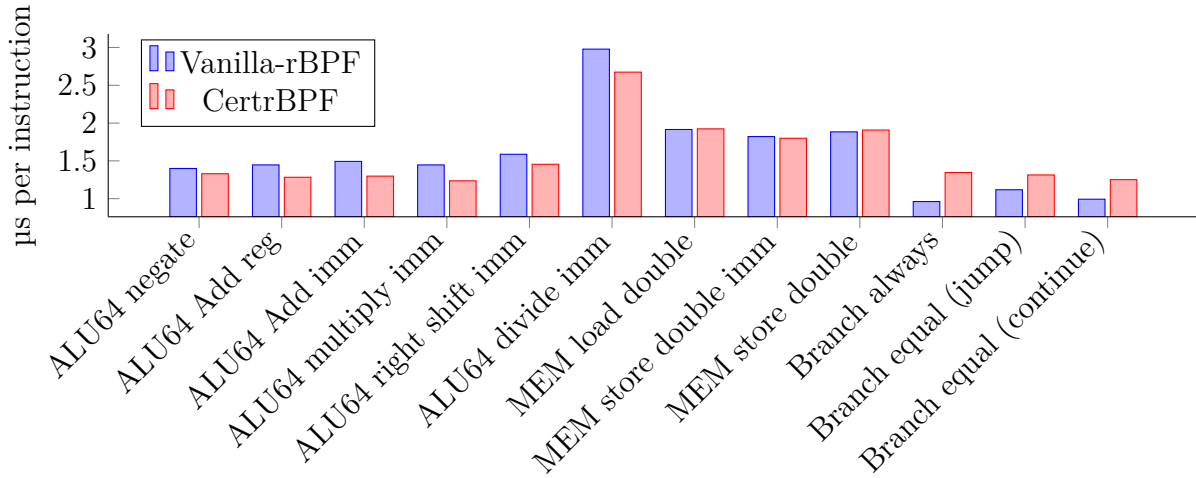


Figure 7.1 – Time per instructions on the Cortex-M4 platform

three branch instructions. These results represent the performance of single instructions averaged over 1000 runs.

Key take-away. In the micro-benchmarks, as depicted in [Figure 7.1](#), most instructions show improved performance with CertrBPF compared to vanilla-rBPF. However, branch instructions exhibit a noticeable slowdown. Possible reasons for this include:

- *Arguments*: When interpreting branch instructions, vanilla-rBPF passes only one argument—a pointer to the current instruction—before computing the opcode, offset, etc., resulting in less execution time. In contrast, CertrBPF first calculates the offset and opcode, which are shared by other instructions, leading to a smaller memory footprint. Then, it copies these values (global monadic state, offset, and opcode) to the branch module.
- *Function calls*: Vanilla-rBPF directly modifies the pointer to a new location, while CertrBPF uses an indirect approach: it accesses the state and modifies the pc field via function calls.

Next, we measure the performance of macro-benchmarks, [Table 7.4](#). The binary programs are compiled using the LLVM-BPF backend (v10.0.0) with default options, *i.e.*, disable ‘-mattr=+alu32’, because both Vanilla-rBPF and CertrBPF support rBPF ALU64 instructions.

The first four benchmarks test pure computation tasks mainly consisting of rBPF Alu64 operations and one extra *exit* instruction (for the purpose of validating the rBPF verifier). These results are averaged over 1000 runs to guarantee accuracy. Then, we select

Interpreter	vanilla-rBPF	CertrBPF	CertrBPF (opt)
incr	8.251 μ s	5.750 μ s	5.376 μs
square	8.251 μ s	5.751 μ s	5.376 μs
bitswap	41.503 μ s	38.628 μ s	38.377 μs
fib	142.001 μ s	151.126 μ s	140.627 μs
sock_buf	195.000 μ s	188.000 μ s	173.000 μs
memcpy_1	25.252 μ s	21.440 μs	23.378 μ s
memcpy_n	683.519 μ s	680.130 μ s	576.749 μs
fletcher32	2959 μ s	2915 μ s	2680 μs
bsort	10 290 μ s	10 253 μ s	9303 μs

Table 7.4 – Execution time of real-world benchmarks (64-bit)

three special cases with more memory operations but fewer `Alu64` operations: the classical BPF socket buffer read/write, memory copies only one element (average over 1000 times), and memory copies many elements. Finally, we benchmark the performance of actual rBPF applications using the Fletcher32 algorithm or the bubble sort algorithm.

Key take-away. In most macro-benchmarks, CertrBPF reduces execution time compared to vanilla-rBPF, as observed in Table 7.4. The exception is the *fib* benchmark, which includes more branch instructions compared to other benchmarks.

Finally, we benchmark the performance of actual IoT data processing, hosted in a femto-container with RIOT-OS running on our selected hardware. In this use case, a sliding window average is performed within the femto-container, on available sensor data points. Figure 7.2 shows the performance measured depending on the size of the window. We use this as blueprint for computation load scaling.

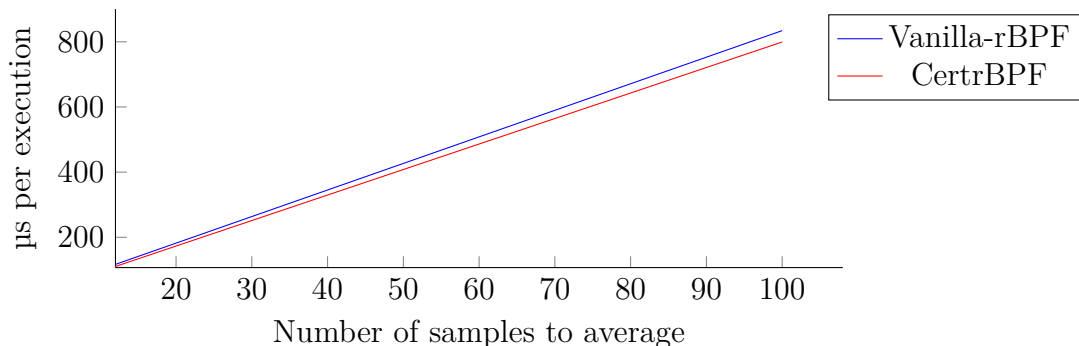


Figure 7.2 – Sliding window average on Cortex-M.

Key take-away. In terms of the sensor data processing benchmark, CertrBPF also decreases the execution time, comparing to vanilla-rBPF, [Figure 7.2](#).

All in all, CertrBPF gains both security and reduces execution time.

7.2.5 Research Question: CertrBPF Interpreter Optimization

RQ3: Does `check_mem` optimization improve the execution of rBPF programs compared to CertrBPF?

We evaluate the performance of macro-benchmarks using CertrBPF, with the `check_mem` optimization both enabled and disabled. As depicted in [Table 7.4](#), the optimized CertrBPF demonstrates improved performance in most macro-benchmarks compared to the original CertrBPF. However, in the case of `memcpy_1`, we observe a slight slowdown in the optimized CertrBPF, although it still outperforms Vanilla-rBPF. This slowdown can be attributed to copying only one element, incurring additional overhead to update the cache but not benefiting from any cache acceleration. On the other hand, the benchmark `memcpy_n` illustrates that the optimized version enjoys a significant speedup thanks to the additional cache.

Discussion. In cases where the optimized CertrBPF exhibits a slowdown, users have the option to disable the `check_mem` optimization. Alternatively, an advanced rBPF verifier could be designed, incorporating static analysis techniques to determine whether the optimization should be enabled or disabled.

7.2.6 Research Question: HAVM Optimization

Our last research question, *RQ4: Does HAVM achieve improved performance compared to CertrBPF and vanilla-rBPF?*

We compare the HAVM implementation against both CertrBPF and Vanilla-rBPF.

Instruction Performance. The raw instruction throughput of the different implementations is measured for a set of different instructions and shown in [Figure 7.3](#). As is visible, in the general case the three implementations have a comparable instruction throughput with some minor differences in performance. Especially performance between HAVM and CertrBPF should be identical for non-JIT instructions. Further investigation showed that the minor differences in performance are caused by non-uniform access to the ROM of the

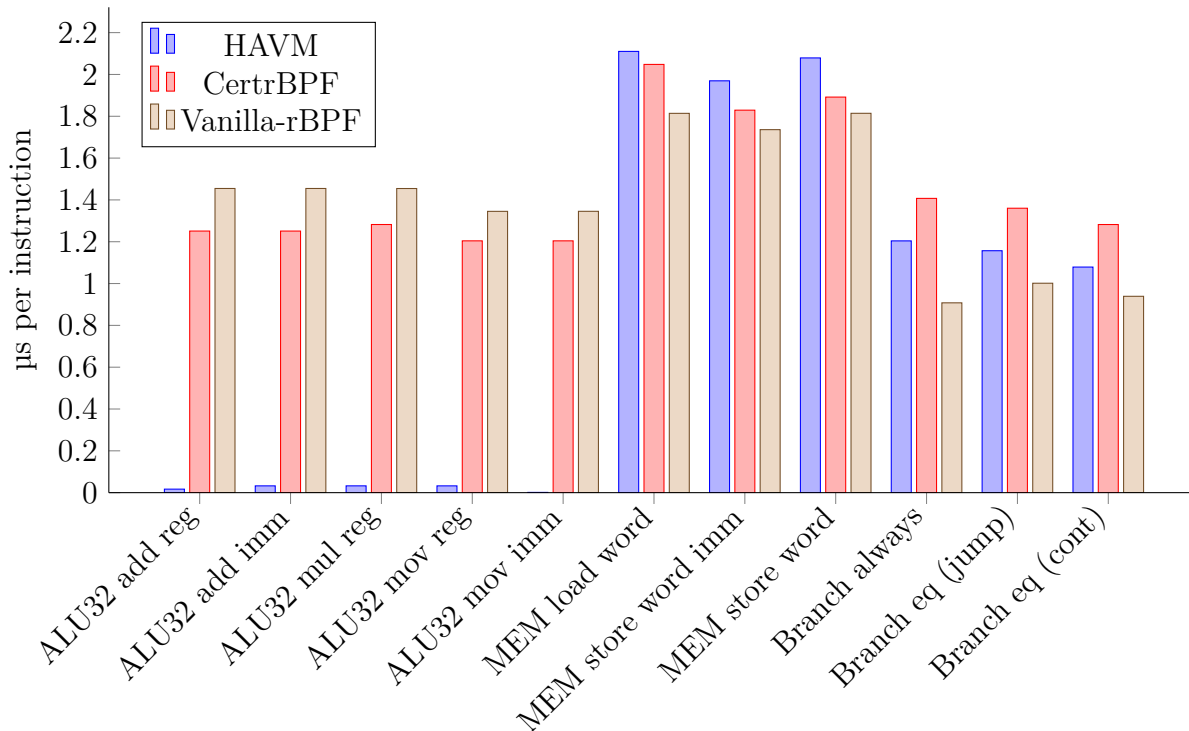


Figure 7.3 – Execution time per instruction, on an Arm Cortex-M4 microcontroller

microcontroller. Depending on the flash layout of the binary by GCC, the performance of the implementations varied slightly.

In the `alu32` case, HAVM is significantly faster than CertrBPF as HAVM uses `JITALU` to accelerate computations. Here, we enable `-mattr=+alu32` so that the LLVM-BPF backend generates only `Alu32` instructions.

When considering the overhead of the JIT compilation process, we measure that the compilation of the instructions incurs a considerable overhead. In [Table 7.5](#) the compilation times and execution times for a number of instructions are shown. As is visible the overhead per compiled instruction varies between $0.7\ \mu\text{s}$ (when no compilation is possible) and $5.1\ \mu\text{s}$. Comparing this to the execution time of the CertrBPF instructions as shown in [Figure 7.3](#), HAVM is always slower when considering the JIT compilation time together with the execution time. However this changes when a script is executed multiple times as the JIT compilation time is an install-time overhead. As soon as a script is executed at least twice, the total install and execution time with HAVM is less than with CertrBPF. We show that this is the case for an rBPF task periodically extracting fine-grained security information/data from the OS kernel or another tracing performance troubleshooting information from a critical application.

Instruction	JIT compilation time	Execution time	Total time
ALU add32 reg	2.69 μ s	0.02 μ s	2.71 μ s
ALU add32 imm	2.08 μ s	0.03 μ s	2.11 μ s
ALU mul32 reg	4.91 μ s	0.06 μ s	4.97 μ s
Non-JIT instructions	0.7 μ s		

Table 7.5 – JIT compilation and execution time of `alu32` instructions

Real World Examples. To show the impact of the JIT compilation with HAVM, we compare the performance between HAVM and CertrBPF using real-world benchmarks, as shown in Table 7.6. These benchmarks use the same source C code as those in Table 7.4, but the corresponding rBPF binary code differs. All ALU instructions are rBPF `Alu32` due to the LLVM BPF backend enabling ‘`-mattr=+alu32`’. We highlight that three worst cases for HAVM: `sock_buf`, `memcpy_1`, and `memcpy_n`. Because they have more memory operations but fewer `Alu32` operations. We observed that, for real-world benchmarks, HAVM improves performance because of `JITALU`: the numerical acceleration feature, compared with CertrBPF.

Interpreter	vanilla-rBPF	CertrBPF	HAVM
<code>incr</code>	8.251 μ s	5.252 μ s	5.000 μs
<code>square</code>	8.251 μ s	5.377 μ s	4.875 μs
<code>bitswap</code>	43.438 μ s	35.128 μ s	18.750 μs
<code>fib</code>	91.503 μ s	89.002 μ s	56.125 μs
<code>sock_buf</code>	330.000 μ s	297.000 μ s	220.000 μs
<code>memcpy_1</code>	43.001 μ s	41.815 μs	44.125 μ s
<code>memcpy_n</code>	864.943 μ s	798.252 μ s	690.000 μs
<code>fletcher32</code>	2214 μ s	1951 μ s	1377 μs
<code>bsort</code>	11 885 μ s	10 696 μ s	8683 μs

Table 7.6 – Execution time of real-world benchmarks (32-bit)

Summary. Our experiments demonstrate that HAVM delivers significantly improved execution times for certain instructions and comparable execution times for others. Notably, `JITALU` in HAVM introduces considerable performance gains. These findings indicate that HAVM offers both speed and versatility.

CONCLUSION

8.1 Summary

Developing real-world operating system libraries in C is a non-trivial task, and verifying the correctness of those implementations is even more complex. When narrowing the context to low-level IoT domains, additional constraints come into play, such as limited memory footprints and higher performance requirements. To address these challenges, this thesis proposes an end-to-end verification workflow and applies it to verify a real-world IoT operating system library, RIOT-OS' rBPF.

[Chapter 4](#) introduces our end-to-end verification workflow that directly derives a verified C implementation from a Gallina specification within the Coq proof assistant. Leveraging the formal semantics of the CompCert C compiler, we establish an end-to-end theorem stating that the final C code inherits the safety and security properties of the Gallina specification. Our approach begins with a monadic proof model in Gallina, which we prove its critical properties in Coq. We then refine this proof model into a synthesis-oriented Gallina model with optimizations. We use an unverified code generator ∂x to extract executable C from the refined model. Importantly, this extraction procedure need not be trusted: it is directly verified using our translation validation theorem.

In [Chapter 5](#), we apply this workflow to generate a verified rBPF virtual machine, the implementation of BPF hosted by the RIOT operating system, from a Gallina specification in Coq. To the best of our knowledge, this is the first verified rBPF virtual machine in the RIOT-OS community. We prove that the rBPF Gallina model satisfies the isolation property, *i.e.*, there are no undefined behaviours in the virtual machine. This isolation property also holds in the verified C implementation due to our end-to-end verification approach. We also introduce the verified `check_mem` optimization to enhance the performance of our verified implementation by using caching to reduce memory accesses.

[Chapter 6](#) discusses the performance aspects of rBPF. We propose a 32-bit variant of rBPF virtual machine whose 32-bit registers and 32-bit instruction operators are naturally

more suitable to be deployed at popular low-power microcontroller architectures. We then develop the first BPF Just-in-Time compiler (JIT_{ALU}) tailored to the hardware and resources constraints of IoT devices, resulting in a refined 32-bit rBPF virtual machine. We integrate JIT_{ALU} with the CertBPF interpreter to create a hybrid virtual machine: HAVM, a defensive, kernel-privileged service capable of accelerating numerical tasks at runtime using partial JIT compilation. The executable C implementation of HAVM is produced by reusing the end-to-end workflow.

In [Chapter 7](#), we demonstrate that all C produces obtained in the aforementioned chapters can be securely integrated into the IoT operating system RIOT-OS to provide the expected functionalities. Preliminary experiments, including benchmarks and an actual IoT data processing, show satisfying memory footprints and performance.

8.2 Perspectives

While we have proposed an end-to-end verification approach and developed a verified RIOT-OS' rBPF virtual machine along with a JIT compiler, we could improve our work in many aspects. In the short term, our goal is to complete a fully verified hybrid virtual machine, HAVM. Additionally, we plan to create a verified ∂x compiler that translates Gallina to CompCert C, which is a long-term goal. We discuss these perspectives in the following two subsections.

8.2.1 Short-term Perspectives

In [Chapter 6](#), we designed and implemented a JIT compiler for rBPF in Coq, marking the first step in our overarching plan. The next step is to fully verify HAVM, including verifying the correctness of our JIT_{ALU} compiler.

- *Backward Design*: We are exploring another JIT design, which we believe may simplify the JIT correctness proof. The current JIT design adopts a ‘forward’ way which is more straightforward to design but harder to prove correct. In this approach, the JIT compiler determines a list of rBPF-32 A1u32 instructions (for Spilling and Save stages) before translating the first element in the rBPF-32 A1u32 list. The new solution employs a ‘backward’ design: the JIT compiler uses information about previous instructions to decide if a particular instruction requires Spilling and Save stages.

-
- *JIT All*: We are also considering a JIT-all compiler that translates all rBPF instructions into ARM binary. The new JIT compiler will adopt the structure introduced in [Section 6.5.2](#) to embed the defensive behaviours into `jited` code. One of the most challenge parts is to embed an optimized `check_mem` algorithm into `jited` code.

8.2.2 Long-term Perspectives

Our end-to-end verification workflow, as discussed in [Chapter 4](#), currently relies on an unverified Gallina code generator, ∂x . We must undertake an additional verification step to ensure the equivalence between the input Gallina model and the extracted C implementation by ∂x . This represents the main limitation of our approach compared to other end-to-end approaches outlined in [Section 2.1](#). To address this limitation, we aim to provide a formally verified Gallina-to-C compiler, akin to CompCert, to thoroughly mitigate this issue. Two potential options are:

- *Translation Validation*. The workflow of ∂x shown in [Figure 3.9](#) relies on an external Coq plugin named Coq-Elpi, and this makes it harder to directly verify the correctness of the ∂x workflow. The translation validation approach doesn't need ∂x to be verified, instead, it requires a verified validator to check the equivalence property between each input Gallina code and the corresponding extracted C program.
- *Forward Simulation Framework (CompCert)*. Another choice is to follow the standard CompCert simulation techniques to create a verified compiler. The strongest competitor running in this track is CertiCoq, a fully verified Compiler (proof underway) that translates arbitrary Gallina programs into executable C code with a verified garbage collector. The main difference between CertiCoq and our expected goal is that we accept a selected Gallina subset as input and generate C code without any garbage collectors.

In contrast to RIOT-OS's rBPF, Linux eBPF boasts a more extensive ecosystem and offers a wider range of application scenarios. We also have plans to extend our verification workflow to ensure the correctness of eBPF's verifier, interpreter, and the various JIT compilers associated with different targets.

BIBLIOGRAPHY

- [Ana+17] Abhishek Anand et al., « CertiCoq : A verified compiler for Coq », *in: CoqPL*, 2017.
- [App+14] Andrew W. Appel et al., *Program logics for certified compilers*, CUP, 2014.
- [Bac+18] Emmanuel Baccelli et al., « RIOT: An open source operating system for low-end embedded devices in the IoT », *in: IoT-J* 5.6 (2018), pp. 4428–4440.
- [BC13] Yves Bertot and Pierre Castéran, *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*, Springer, 2013.
- [BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy, « Formal Verification of a C Compiler Front-End », *in: FM 2006: Formal Methods*, ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 460–475, ISBN: 978-3-540-37216-5.
- [Ber07] Gérard Berry, « SCADE: Synchronous design and validation of embedded control software », *in: Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems: Proceedings of the GM R&D Workshop, Bangalore, India, January 2007*, Springer, 2007, pp. 19–33.
- [BL09] Sandrine Blazy and Xavier Leroy, « Mechanized Semantics for the Clight Subset of the C Language », *in: Journal of Automated Reasoning* 43 (2009), pp. 263–288, URL: <https://api.semanticscholar.org/CorpusID:896527>.
- [Bou+17] Timothy Bourke et al., « A Formally Verified Compiler for Lustre », *in: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain: Association for Computing Machinery*, 2017, pp. 586–601, ISBN: 9781450349888, DOI: [10.1145/3062341.3062358](https://doi.org/10.1145/3062341.3062358), URL: <https://doi.org/10.1145/3062341.3062358>.

-
- [Cas+87] P. Caspi et al., « LUSTRE: A Declarative Language for Real-Time Programming », *in: Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, Munich, West Germany: Association for Computing Machinery, 1987, pp. 178–188, ISBN: 0897912152, DOI: [10.1145/41625.41641](https://doi.org/10.1145/41625.41641), URL: <https://doi.org/10.1145/41625.41641>.
- [Cer23a] the CertrBPF team, *the CertrBPF repo*, 2023, URL: <https://gitlab.inria.fr/syuan/rbpf-dx/-/tree/CAV22-AE/>.
- [Cer23b] the CertrBPF team, *the CertrBPFOpt repo*, 2023, URL: <https://gitlab.inria.fr/syuan/certrbpfopt/-/tree/main>.
- [Coh+09] Ernie Cohen et al., « VCC: A practical system for verifying concurrent C », *in: International Conference on Theorem Proving in Higher Order Logics*, Springer, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 23–42.
- [Cou+05] Patrick Cousot et al., « The ASTREÉ Analyzer », *in: Programming Languages and Systems*, ed. by Mooly Sagiv, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 21–30, ISBN: 978-3-540-31987-0.
- [DB08] 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, Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340, ISBN: 3540787992.
- [Del00] David Delahaye, « A Tactic Language for the System Coq », *in: Logic for Programming and Automated Reasoning*, ed. by Michel Parigot and Andrei Voronkov, Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 85–95, ISBN: 978-3-540-44404-6.
- [Dun+15] Cvetan Dunchev et al., « ELPI: Fast, Embeddable, λ Prolog Interpreter », *in: LPAR*, vol. 9450, LNCS, Springer, 2015, pp. 460–468.
- [Erb+21] Andres Erbsen et al., « Integration Verification across Software and Hardware for a Simple Embedded System », *in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, Virtual, Canada: Association for Computing

-
- Machinery, 2021, pp. 604–619, ISBN: 9781450383912, DOI: [10.1145/3453483.3454065](https://doi.org/10.1145/3453483.3454065), URL: <https://doi.org/10.1145/3453483.3454065>.
- [Fle17] Matt Fleming, « A Thorough Introduction to eBPF », *in: Linux Weekly News* (2017).
- [Gef+20] Jacob Van Geffen et al., « Synthesizing jit compilers for in-kernel dsls », *in: International Conference on Computer Aided Verification*, Springer, 2020, pp. 564–586.
- [Ger+19] Elazar Gershuni et al., « Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions », *in: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*, Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 1069–1084, ISBN: 9781450367127, DOI: [10.1145/3314221.3314590](https://doi.org/10.1145/3314221.3314590), URL: <https://doi.org/10.1145/3314221.3314590>.
- [Gu+16] Ronghui Gu et al., « CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. », *in: OSDI, USENIX*, 2016, pp. 653–669.
- [Haw+14] Chris Hawblitzel et al., « Ironclad Apps: End-to-End Security via Automated Full-System Verification », *in: Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation, OSDI'14*, Broomfield, CO: USENIX Association, 2014, pp. 165–181, ISBN: 9781931971164.
- [Hue92] Gérard Huet, « The Gallina specification language: A case study », *in: Foundations of Software Technology and Theoretical Computer Science*, ed. by Rudrapatna Shyamasundar, Berlin, Heidelberg: Springer Berlin Heidelberg, 1992, pp. 229–240, ISBN: 978-3-540-47507-1.
- [Jom+18a] Narjes Jomaa et al., « Formal proof of dynamic memory isolation based on MMU », *in: Science of Computer Programming* 162 (2018), Special Issue on TASE 2016, pp. 76–92, ISSN: 0167-6423, DOI: <https://doi.org/10.1016/j.scico.2017.06.012>, URL: <https://www.sciencedirect.com/science/article/pii/S0167642317301338>.
- [Jom+18b] Narjes Jomaa et al., « Proof-Oriented Design of a Separation Kernel with Minimal Trusted Computing Base », *in: AVOCS*, vol. 76, Electronic Communications of the EASST, 2018.

-
- [JP08] Bart Jacobs and Frank Piessens, *The VeriFast program verifier*, tech. rep., Citeseer, 2008.
- [JPL12] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy, « Validating LR(1) Parsers », *in: Programming Languages and Systems*, ed. by Helmut Seidl, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 397–416, ISBN: 978-3-642-28869-2.
- [Kir+15] Florent Kirchner et al., « Frama-C: A software analysis perspective », *in: Formal Aspects of Computing 27.3* (May 2015), pp. 573–609, DOI: [10.1007/s00165-014-0326-7](https://doi.org/10.1007/s00165-014-0326-7), URL: <https://cea.hal.science/cea-01808981>.
- [Kle+09] Gerwin Klein et al., « seL4: formal verification of an OS kernel », en, *in: SOSp*, ACM Press, 2009, p. 207, ISBN: 978-1-60558-752-3, (visited on 12/16/2021).
- [KR02] Brian W Kernighan and Dennis M Ritchie, « The C programming language », *in:* (2002).
- [LB08] Xavier Leroy and Sandrine Blazy, « Formal verification of a C-like memory model and its uses for verifying program transformations », *in: JAR 41.1* (2008), pp. 1–31.
- [Lei10] K. Rustan M. Leino, « Dafny: An Automatic Program Verifier for Functional Correctness », *in: Logic for Programming, Artificial Intelligence, and Reasoning*, ed. by Edmund M. Clarke and Andrei Voronkov, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370, ISBN: 978-3-642-17511-4.
- [Ler+12] Xavier Leroy et al., *The CompCert Memory Model, Version 2*, Research Report RR-7987, INRIA, 2012, p. 26.
- [Ler09] Xavier Leroy, « Formal verification of a realistic compiler », en, *in: Communications of the ACM 52.7* (2009), pp. 107–115, ISSN: 0001-0782, 1557-7317, (visited on 12/16/2021).
- [Let02] Pierre Letouzey, « A new extraction for Coq », *in: TYPES*, Springer, 2002, pp. 200–219.
- [MJ93] Steven McCanne and Van Jacobson, « The BSD Packet Filter: A New Architecture for User-level Packet Capture », *in: Usenix Winter Conference*, vol. 46, USENIX, 1993, pp. 259–270.
- [MK14] Nicholas D Matsakis and Felix S Klock, « The rust language », *in: ACM SIGAda Ada Letters 34.3* (2014), pp. 103–104.

-
- [MO12] Magnus O. Myreen and Scott Owens, « Proof-Producing Synthesis of ML from Higher-Order Logic », *in: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming, ICFP '12*, Copenhagen, Denmark: Association for Computing Machinery, 2012, pp. 115–126, ISBN: 9781450310543, DOI: [10.1145/2364527.2364545](https://doi.org/10.1145/2364527.2364545), URL: <https://doi.org/10.1145/2364527.2364545>.
- [Mou+15] Leonardo de Moura et al., « The Lean Theorem Prover (System Description) », *in: Automated Deduction - CADE-25*, ed. by Amy P. Felty and Aart Middeldorp, Cham: Springer International Publishing, 2015, pp. 378–388, ISBN: 978-3-319-21401-6.
- [Mul+18] Eric Mullen et al., « Cεuf: minimizing the Coq extraction TCB », *in: CPP*, ACM, 2018, pp. 172–185.
- [Nel+17] Luke Nelson et al., « Hyperkernel: Push-Button Verification of an OS Kernel », *in: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17*, Shanghai, China: Association for Computing Machinery, 2017, pp. 252–269, ISBN: 9781450350853, DOI: [10.1145/3132747.3132748](https://doi.org/10.1145/3132747.3132748), URL: <https://doi.org/10.1145/3132747.3132748>.
- [Nel+19] Luke Nelson et al., « Scaling Symbolic Evaluation for Automated Verification of Systems Code with Serval », *in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP '19*, Huntsville, Ontario, Canada: Association for Computing Machinery, 2019, pp. 225–242, ISBN: 9781450368735, DOI: [10.1145/3341301.3359641](https://doi.org/10.1145/3341301.3359641), URL: <https://doi.org/10.1145/3341301.3359641>.
- [Nel+20] Luke Nelson et al., « Specification and verification in the field: Applying formal methods to {BPF} just-in-time compilers in the Linux kernel », *in: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 41–61.
- [NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, Berlin, Heidelberg: Springer-Verlag, 2002, ISBN: 3540433767.
- [Pit+22] Clément Pit-Claudel et al., « Relational Compilation for Performance-Critical Applications », *in: PLDI*, ACM, 2022.

-
- [Pro+17] Jonathan Protzenko et al., « Verified Low-Level Programming Embedded in F* », *in: PACMPL 1.ICFP* (Sept. 2017), 17:1–17:29, DOI: [10.1145/3110261](https://doi.org/10.1145/3110261).
- [PSS98] Amir Pnueli, Michael Siegel, and Eli Singerman, « Translation Validation », *in: TACAS*, ed. by Bernhard Steffen, vol. 1384, LNCS, Springer, 1998, pp. 151–166.
- [Rie17] Leanna Rierson, *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance*, CRC Press, 2017.
- [Riz+16] Christine Rizkallah et al., « A Framework for the Automatic Formal Verification of Refinement from Cogent to C », *in: ITP*, vol. 9807, LNCS, Springer, 2016, pp. 323–340.
- [RL10] Silvain Rideau and Xavier Leroy, « Validating Register Allocation and Spilling », *in: Compiler Construction*, ed. by Rajiv Gupta, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 224–243, ISBN: 978-3-642-11970-5.
- [Sam+21] Michael Sammler et al., « RefinedC: Automating the Foundational Verification of C Code with Refined Ownership Types », *in: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, New York, NY, USA: Association for Computing Machinery, 2021, pp. 158–174, ISBN: 9781450383912, URL: <https://doi.org/10.1145/3453483.3454036>.
- [SH23] Bhat Sanjit and Shacham Hovav, *Formal Verification of the Linux Kernel eBPF Verifier Range Analysis*, 2023, URL: <https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf>.
- [Soz+20] Matthieu Sozeau et al., « The MetaCoq Project », *in: Journal of Automated Reasoning* (Feb. 2020), DOI: [10.1007/s10817-019-09540-0](https://doi.org/10.1007/s10817-019-09540-0), URL: <https://inria.hal.science/hal-02167423>.
- [Swa+13] Nikhil Swamy et al., « Verifying Higher-order Programs with the Dijkstra Monad », *in: Proceedings of the 34th annual ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '13, 2013, pp. 387–398.
- [Tan21] Akira Tanaka, « Coq to C translation with partial evaluation », *in: PEPM@POPL*, ACM, 2021, pp. 14–31.

-
- [Tas21] Enrico Tassi, *Coq-Elpi, Coq plugin embedding Elpi*, 2021, URL: <https://github.com/LPCIC/coq-elpi>.
- [TB14] Emina Torlak and Rastislav Bodik, « A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages », *in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, Edinburgh, United Kingdom: Association for Computing Machinery, 2014, pp. 530–541, ISBN: 9781450327848, DOI: [10.1145/2594291.2594340](https://doi.org/10.1145/2594291.2594340), URL: <https://doi.org/10.1145/2594291.2594340>.
- [Vis+23] Harishankar Vishwanathan et al., « Verifying the Verifier: eBPF Range Analysis Verification », *in: Computer Aided Verification*, ed. by Constantin Enea and Akash Lal, Cham: Springer Nature Switzerland, 2023, pp. 226–251, ISBN: 978-3-031-37709-9.
- [Wan+14] Xi Wang et al., « Jitk: A Trustworthy In-Kernel Interpreter Infrastructure », *in: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, 2014, pp. 33–47.
- [Xu+16] Fengwei Xu et al., « A Practical Verification Framework for Preemptive OS Kernels », *in: Computer Aided Verification*, ed. by Swarat Chaudhuri and Azadeh Farzan, Cham: Springer International Publishing, 2016, pp. 59–79, ISBN: 978-3-319-41540-6.
- [YT21] Shenghao Yuan and Jean-Pierre Talpin, « Verified Functional Programming of an IoT Operating System’s Bootloader », *in: Proceedings of the 19th ACM-IEEE International Conference on Formal Methods and Models for System Design, MEMOCODE '21*, Virtual Event, China: Association for Computing Machinery, 2021, pp. 89–97, ISBN: 9781450391276, DOI: [10.1145/3487212.3487347](https://doi.org/10.1145/3487212.3487347), URL: <https://doi.org/10.1145/3487212.3487347>.
- [Yua+22] Shenghao Yuan et al., « End-to-End Mechanized Proof of an eBPF Virtual Machine for Micro-controllers », *in: Computer Aided Verification*, ed. by Sharon Shoham and Yakir Vizel, Cham: Springer International Publishing, 2022, pp. 293–316.
- [ZB20] Koen Zandberg and Emmanuel Baccelli, « Minimal Virtual Machines on IoT Microcontrollers: The Case of Berkeley Packet Filters with rBPF », *in: PEMWN, IEEE*, 2020, pp. 1–6.

-
- [Zha+13] Jianzhou Zhao et al., « Formal Verification of SSA-Based Optimizations for LLVM », in: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 175–186, ISBN: 9781450320146, DOI: [10.1145/2491956.2462164](https://doi.org/10.1145/2491956.2462164), URL: <https://doi.org/10.1145/2491956.2462164>.
- [ZS19] Yongwang Zhao and David Sanán, « Rely-Guarantee Reasoning About Concurrent Memory Management in Zephyr RTOS », in: *Computer Aided Verification*, ed. by Isil Dillig and Serdar Tasiran, Cham: Springer International Publishing, 2019, pp. 515–533, ISBN: 978-3-030-25543-5.

Titre : Programmation vérifiée et intégration sécurisée de bibliothèques de systèmes d'exploitation dans Coq

Mot clés : BPF, Coq, Vérification formelle, Génération de code

Résumé : En tant que technologie révolutionnaire d'extension du noyau, Berkeley Packet Filters (BPF) a été appliqué à divers systèmes d'exploitation dans différents domaines, des serveurs (BPF étendu de Linux) aux microcontrôleurs (rBPF de RIOT-OS). L'isolation des machines virtuelles BPF est essentielle pour garantir l'intégrité du système contre les programmes potentiellement malveillants, en particulier pour les microcontrôleurs qui disposent rarement d'une protection matérielle de la mé-

moire. Cette thèse présente une machine virtuelle rBPF de confiance dont l'isolation des fautes est formellement prouvée dans l'assistant de preuve Coq. Nous présentons un processus de vérification de bout en bout pour extraire une implémentation C exécutable vérifiée à partir de modèles rBPF abstraits écrits en Coq. Nous introduisons également des techniques Just-in-Time dans rBPF pour l'optimisation des performances. Nos preuves sont toutes vérifiées mécaniquement dans Coq.

Title: Verified programming and secure integration of operating system libraries in Coq

Keywords: BPF, Coq, Formal Verification, Code Generation

Abstract: As a revolutionary kernel extension technology, Berkeley Packet Filters (BPF) has been applied for various operating systems from different domains, from servers (Linux's extended BPF) to micro-controllers (RIOT-OS rBPF). The isolation of BPF virtual machines (VM) is critical to ensure system integrity against potentially malicious programs, especially for micro-controllers that rarely feature hardware memory protection. This the-

sis presents a trusted rBPF virtual machine that is formally proven fault-isolate in the Coq proof assistant. We present an end-to-end verification workflow for extracting verified executable C implementation from abstract rBPF models written in Coq. We also introduce Just-in-Time techniques into rBPF for performance optimization. All our proofs have been mechanized in Coq.