Hardware security for just-in-time compilation in language virtual machines

Quentin Ducasse

COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

ENSTA BRETAGNE

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
DE TECHNIQUES AVANCÉES BRETAGNE

ÉCOLE DOCTORALE N° 648
Sciences pour l'Ingénieur et le Numérique
Spécialité : *STIC - Informatique*

Par

Quentin DUCASSE

Sécurisation matérielle de la compilation à la volée des machines virtuelles langage

Thèse présentée et soutenue à Brest, le vendredi 29 mars 2024
Unité de recherche : Lab-STICC, équipe ARCAD

Rapporteurs avant soutenance :

Sébastien PILLEMENT	Professeur - Université de Nantes
Laure GONNORD	Professeure - Grenoble INP / Esisar

Composition du Jury :

Président :	Ludovic MÉ	Professeur - INRIA
Examinateurs :	Gaël THOMAS	Télécom SudParis
	Zoé DREY	Ingénieure
Dir. de thèse :	Loïc LAGADEC	Professeur - ENSTA Bretagne
Co-encadrant de thèse :	Pascal COTRET	Enseignant-chercheur - ENSTA Bretagne

Invité :

Guillermo POLITO	Chargé de recherche - INRIA Lille

Remerciements

Je tiens tout d'abord à remercier mon encadrant, Pascal Cotret, pour le soutien qu'il a pu m'apporter ainsi que les connaissances qu'il m'a transmises durant la réalisation de ces travaux. Je tiens également à remercier mon directeur de thèse, Loïc Lagadec, pour les conseils avisés qu'il m'a prodigués tant dans la direction des travaux de recherche que dans la qualité de ses remarques. Je leur suis extrêmement reconnaissant de la confiance qu'ils m'ont accordée, laquelle m'a permis de m'épanouir scientifiquement à leurs côtés.

J'adresse toute ma reconnaissance à Sébastien Pillement et Laure Gonnord qui m'ont fait l'honneur d'accepter d'évaluer mes travaux en tant que rapporteurs. Je remercie également Ludovic Mé, Gaël Thomas et Zoé Drey d'avoir accepté de faire partie du jury d'évaluation. Je souhaite ensuite remercier Erven Rohou et Lilian Bossuet d'avoir participé à mon comité de suivi de thèse et pour leurs conseils.

Je remercie également Guillermo Polito et Pablo Tesone pour m'avoir accueilli et épaulé pendant mon échange au sein de l'équipe RMoD. Je les remercie pour le soutien qu'ils m'ont apporté ainsi que la qualité de nos échanges techniques et scientifiques.

Je suis également reconnaissant d'avoir pu travailler au sein de l'ENSTA Bretagne où j'ai eu la chance de côtoyer et travailler avec des collègues et amis qui m'ont soutenu et accompagné dans mes activités d'enseignement et de recherche.

Enfin, je remercie chaleureusement mes proches, mes amis et ma famille pour leur soutien permanent et leur confiance au long de ces années où j'ai pu être moins disponible pour eux. Je remercie particulièrement mes parents pour leur enthousiasme communicatif ainsi qu'Armelle Gillot pour son soutien constant et sa bienveillance.

Structure

Note sur la structure de la thèse

Cette thèse est écrite en anglais pour ses chapitres principaux d'état de l'art, des contributions, ainsi que l'introduction et la conclusion. Un résumé en français reprenant les points principaux de chacun des chapitres est présenté en français après la table des matières. Un résumé plus court en français et en anglais se trouve en quatrième de couverture.

Note on the thesis structure

This thesis is written in English for its main chapters of state-of-the-art, contributions, as well as the introduction and conclusion. A summary in French of the main points of each chapter is presented after the table of contents. A shorter summary in French and English can be found on the back cover.

Table of Contents

List of Tables

List of Figures

Résumé de la thèse

Introduction

Ces dernières décennies ont vu la sécurité d'un nombre croissant de systèmes d'information compromise, faisant fuiter les données personnelles et confidentielles de millions d'utilisateurs. En 2014, la vulnérabilité "Heartbleed" [1] affecte entre 24 et 55% des sites HTTPS les plus utilisés. Ce bug de la librairie de cryptographie OpenSSL [2], très largement distribuée, utilisait les données envoyées par un utilisateur sans vérification préalable, lui permettant d'exfiltrer des informations confidentielles stockées par le serveur. Pour garantir l'intégrité et la sécurité des données d'un utilisateur d'un système d'information, plusieurs solutions sont mises en place au niveau de l'architecture matérielle directement, du système d'exploitation (OS), ou à travers des environnements d'exécution dédiés.

Parmi ces environnements d'exécution, les machines virtuelles langage (VMs) supportent l'exécution d'un langage de programmation en faisant abstraction du processus de compilation et d'allocation mémoire pour l'utilisateur. Elles permettent la portabilité directe du code applicatif sur les architectures supportées par la VM. Java ou Python utilisent chacun une VM pour leur exécution. Ce sont des logiciels complexes qui gèrent à la fois la compilation du langage source, son optimisation, et l'utilisation de la mémoire par l'application. Les caractéristiques de portabilité et de garanties d'exécution font des VMs et des langages qu'elles supportent des outils de développement intéressants. Des VMs sont déployées sur la majorité des systèmes d'information, à travers par exemple un navigateur web et sa VM JavaScript chargée d'exécuter sur l'ordinateur de l'utilisateur le code JavaScript fourni par un serveur dans une page web. Un autre exemple, l'Android Run Time (ART) permet l'exécution d'applications Android et est déployée sur tous les smartphones utilisant cet OS. Le déploiement et l'adoption globale de VMs sur une variété de systèmes d'information à usage du grand public, couplé à leur capacité à exécuter des tâches à criticité importante telles que la manipulation de la mémoire et la génération de code machine en font des cibles intéressantes pour un attaquant. En particulier, le composant chargé de la compilation à la volée ("just-in-time" ou JIT en anglais), est critique à la performance et manipule du code exécutable.

Nous nous intéressons dans le contexte de cette thèse à la mise en place de défenses autour des VMs, et en particulier à la sécurisation du code recompilé à la volée. Nous motivons aussi la pertinence de l'accélération matérielle de ces défenses dans un objectif de co-design applicatif/matériel. Nous présentons trois contributions principales :

Un générateur de binaires instrumentés, Gigue. Il génère des exécutables ressemblant à des régions de code machine recompilé par la VM, selon des paramètres de caractérisation. Les binaires sont complétés par des instructions aléatoires selon la caractérisation, et instrumentables avec des instructions dédiées. Ils sont directement exécutables sur un processeur synthétisable ou son simulateur associé. Gigue permet la génération de binaires variés et d'un socle de comparaison de solutions de sécurité.

Une solution d'isolation par les instructions, JITDomain. Cette solution d'isolation définit des domaines à travers des instructions dédiées. La solution est appliquée au code machine recompilé par une VM. Elle permet la garantie d'aspects importants de sécurité à l'exécution comme la séparation de l'accès aux données, l'isolation de la pile d'appel et le filtrage des appels systèmes. La solution est déployée sur le processeur RISC-V open-source CVA6 [3] avec un coût minimal en ressources matérielles et performance du binaire instrumenté.

Le port d'un compilateur JIT sur RISC-V, Cogit le compilateur à la volée de la VM Pharo. Nous présentons le port des outils de développement et de tests utilisés par la VM Pharo sur l'architecture RISC-V. Le compilateur JIT utilise une représentation intermédiaire qui n'est pas directement interfaçable avec le jeu d'instruction RISC-V. Le port valide l'intégralité des tests unitaires mis en place par la VM Pharo et l'utilisation du compilateur JIT permet une accélération du nombre de bytecode et d'appels de fonction exécutés. Nous étendons les outils de simulation de la VM pour permettre l'exécution d'instructions personnalisées.

Nous présentons dans la suite de ce résumé une partie de l'état de l'art sur les attaques et défenses de VMs ainsi que la présentation de solutions de sécurité matérielle liées à l'isolation fine. Nous présentons ensuite un résumé de chacune des trois contributions, puis les perspectives qui résultent de ces travaux.

État de l'Art

Les machines virtuelles langage (VMs) sont des environnements d'exécution complexes qui contiennent différents niveaux de compilation et d'exécution. Le code source est d'abord traduit sous forme d'un arbre de syntaxe abstraite (*abstract syntax tree* ou AST). Cet arbre est ensuite transformé en une autre représentation intermédiaire, propre à l'interprétation : le *bytecode*. L'interpréteur exécute la succession de bytecode directement, un à un via un "*switch*" sur toutes les instructions existantes. Ce processus amène à une exécution ralentie du code applicatif par rapport à du code directement compilé. Lorsqu'une suite de bytecode est souvent utilisée, la VM fait appel à un ou plusieurs compilateur(s) à la volée (ou "*just-in-time*" (*JIT*) en anglais) qui recompilent cette succession de bytecode en code machine directement. La granularité et la méthode de recompilation dépendent de la VM et de son implémentation. Une trace d'exécution ou une fonction fréquemment appelée peuvent être sélectionnés et recompilés. Enfin, ces composants qui contrôlent le flot d'exécution travaillent avec le ramasse-miettes (ou "*garbage collector*" (*GC*) en anglais) pour attribuer et récupérer de la mémoire. Plusieurs algorithmes et méthodes d'attribution et récupération de mémoire sont utilisés pour minimiser la latence et le temps passé à récupérer la mémoire inutilisée. Ces composants sont responsables de l'exécution et la gestion du code applicatif du langage qu'ils supportent.

Dans l'ensemble, les attaques contre les VM se sont d'abord concentrées sur le compilateur JIT et la région de code JIT. Celle-ci nécessite un accès en écriture pour y placer le code machine ainsi qu'un accès en exécution pour rediriger le flot de contrôle vers la nouvelle version du code. Ces caractéristiques en ont fait un composant sensible à l'injection de code [4]. La zone de code JIT peut être utilisée pour mettre en défaut des mesures comme la distribution aléatoire de l'espace d'adressage (ou *ASLR* en anglais) en utilisant la prédictibilité de la génération du code. D'autres attaques se servent du compilateur JIT pour renforcer des attaques existantes de réutilisation de code [5], voire l'injection de code lui-même propice à la réutilisation de code [6]-[8]. La réutilisation de code se base sur la détection de "*gadgets*", des petites successions de code machines dédiées à une tâche particulière et dont l'enchaînement permet d'exécuter un code arbitraire. Enfin, les représentations intermédiaires que la VM utilise sont aussi des cibles potentielles d'injections malicieuses. C'est le cas pour le bytecode [9], instructions que l'interpréteur consomme, ou la représentation intermédiaire du compilateur JIT pour la recompilation [10]. Ces trois attaques se caractérisent en "*code injection*", "*code reuse*" et "*data-only*".

Au fur et à mesure de l'évolution de la sécurité des logiciels, les défenses autour des machines virtuelles se sont d'abord concentrées sur la perturbation du déterminisme introduit par le compilateur JIT. Des propriétés telles que la diversité et la randomisation jouent un rôle essentiel dans le renforcement de la sécurité du processus de compilation JIT, en particulier contre les simples attaques par injection [11]-[13]. Toutefois, ces mesures peuvent s'avérer inadéquates face à des menaces plus sophistiquées ou pour assurer des garanties plus fortes. Le contrôle des permissions de mémoire est une méthode simple pour se prémunir contre l'injection de code, en empêchant les attaquants d'écrire dans la mémoire exécutable [14], [15]. Étant donné que le compilateur JIT a explicitement besoin de mémoire exécutable, la double allocation d'une région mémoire avec des permissions différentes permet une séparation des accès [16], [17]. La réutilisation du code reste un problème, le code JIT exposant les gadgets disponibles aux attaquants. Deux mesures de protection consistent à supprimer la lisibilité du code JIT exécutable et à mettre en œuvre un mécanisme robuste d'intégrité du flot de contrôle [18], [19]. Une isolation restrictive à grain fin entre les composants offre la plus grande fiabilité à la VM, mais a un impact net sur les performances, impact amorti à travers une accélération matérielle [9], [10].

Les solutions d'isolation fine à l'exécution sont coûteuses lors de leur exécution complète du côté logiciel. Une proposition de primitives de sécurité supportées matériellement amortit ce coût en performance. Pour pouvoir expérimenter avec le design et l'implémentation de telles solutions, RISC-V [20] est un jeu d'instruction open-source, modulaire et qui permet l'ajout de nouvelles instructions dédiées. Les solutions d'isolation fines se séparent en trois catégories importantes. Tout d'abord, l'accélération de solutions d'intégrité de flot de contrôle. Intel supporte une pile d'appels dupliquée et séparée dans l'espace d'adressage avec CET [21, Vol.1 Ch.17]. Des solutions comparables sont disponibles sur RISC-V [22], [23] et proposent une vérification du flot de contrôle accélérée. Ensuite, le marquage et la séparation de la mémoire à travers l'extension des pointeurs, qu'ARM définit avec MTE [24] et PAC [25, Ch.B6]. RISC-V propose un équivalent sur les adresses de la pile d'appel [26] ou l'ajout de tags en mémoire [27]. Ces pointeurs peuvent être étendus avec de nombreuses métadonnées utilisées à l'exécution [28]-[30]. Enfin, l'isolation de domaines via un moniteur dédié et l'extension des pages de mémoire [31], [32] étend le principe des clés mémoire mis en place par Intel MPK [21, Vol.3 Sec.4.6]. Cette isolation peut aussi se mettre en place directement par l'attribution d'un domaine à une instruction spécifique [33], [34]. Ces solutions sont légères à la mise en place et accélérées matériellement pour garantir une isolation fine

Gigue : Un Générateur de Code JIT Instrumentable

La comparaison de solutions d'isolation dédiées au code JIT nécessite un changement à trois niveaux applicatifs : l'intégration dans un processeur, une interface du système d'exploitation ainsi qu'une extension de la VM. Cette pile technologique ralentit considérablement l'expérimentation, en particulier pour le développeur matériel qui a un cycle de retour d'expérience de son implémentation allongé. Pour permettre une possibilité de co-design logiciel/matériel, nous proposons Gigue, un générateur de binaires similaire à la région de code JIT qui propose une interface pour instrumenter les binaires à différents niveaux. Notre objectif est de pouvoir exécuter ces binaires directement sur les simulateurs "*cycle-accurate*" de processeurs RISC-V open-source.

Gigue est construit autour de trois objectifs principaux : la paramétrisation et détermination de caractéristiques des binaires générés, la modularité de sa génération permettant l'instrumentation des binaires à différents niveaux, ainsi que la vérification de l'exécution des binaires avant leur distribution. Les binaires que Gigue génère sont compilé statiquement et donc auto-contenus dans un exécutable `ELF`. Leur structure est inspirée de la région de code JIT de la VM Pharo. Ils sont composés d'une région qui contient différentes méthodes JIT remplies aléatoirement d'instructions, ainsi que d'autres éléments tels que des caches polymorphiques [35], ou des "trampolines" qui contrôlent le flot d'exécution venant et sortant. Une autre région appelle successivement toutes les méthodes présentes dans la première dans un ordre aléatoire. Pour générer le binaire correspondant, toutes les méthodes sont instanciées, remplies d'instructions, puis Gigue en déduit un graphe de flot de contrôle et ajoute des appels inter-fonctions dans le corps des méthodes. Il compacte les différentes parties dans un binaire directement exécutable.

De nombreux paramètres permettent de contrôler la génération de binaires de Gigue. Tout d'abord, la caractérisation de la zone contenant les méthodes JIT à travers sa taille et la fréquence d'apparition de ses différents éléments. Ensuite, le type d'application généré est contrôlée par la caractérisation des méthodes elles-même (taille et variation de taille), la distribution des types d'instruction qui les compose (arithmétique, accès mémoire, changement de flot de contrôle) ou la fréquence d'utilisation des appels ainsi que leur profondeur. Ces paramètres permettent de générer et amplifier des classes d'applications données pour tester différents paramètres. Dans un cas d'usage, nous utilisons des informations d'exécution de différents benchmarks [36], [37] ainsi que des informations sur la région de code JIT Pharo [38]. Nous dérivons 9 classes d'applications avec des

tailles de méthodes et densité d'appels différentes. De manière similaire, nous générons 9 classes d'applications différentes en variant la taille des méthodes et la quantité d'appels mémoire qui compose les méthodes. Nous démontrons l'interfaçage direct de Gigue avec deux processeurs applicatifs open-source : CVA6 [3] et Rocket [39].

Pour permettre l'ajout d'instructions "*custom*" au sein des binaires générés par Gigue, nous définissons plusieurs points d'accroche dans le processus de génération. Trois exemples sont intégrés dans Gigue, un premier ajoute des instructions de rotation qui ne sont pas ajoutées au jeu d'instruction par défaut ; un second ajoute une "*shadow stack*", une pile d'appel dupliquée qui stocke les adresses de retour mises en place lors d'un appel de fonction ; enfin, une authentification de pointeur est mise en place, authentifiant les adresses de retour des fonctions lors de l'appel et les vérifiant lors du retour du flot de contrôle. Ces trois solutions sont ajoutées dans les binaires soit dans le corps des méthodes (permier cas) ou dans le prologue et l'épilogue des fonctions. Pour vérifier leur implémentation dans le binaire, nous proposons une interface de test simplifié qui utilise le simulateur de processeur Unicorn [40]. Les nouvelles instructions sont détectées, et lors de la levée d'une exception les concernant, la routine correspondante est exécutée sur l'état du processeur avant de continuer l'exécution. Ce support permet de garantir l'exécution des binaires avant leur distribution, ainsi que de vérifier l'intégration des instructions dans le binaire.

JITDomain : Sécurisation du Code JIT par les Instructions

Extraits de l'état de l'art et des solutions de sécurité déjà déployées autour des VMs et leurs éléments, nous définissons plusieurs garanties nécessaires à la sécurisation du code JIT. Tout d'abord, la séparation de la pile d'appel et la vérification des adresses de retour permettent de garantir l'intégrité du flot de contrôle à gros grain. Ensuite, l'accès des données contenues dans la région du code JIT doit être strictement limitée aux accès provenant du code JIT lui-même. Enfin, le filtrage des appels système restreint les capacités d'un attaquant qui contrôlerait la région du code JIT. Pour permettre leur adoption, la mise en place de ces garanties doit se faire avec un impact minimum sur les ressources matérielles du processeur ainsi qu'un impact minimum sur la performance et la taille du code JIT instrumenté.

Afin de séparer la région du code JIT du reste de l'application, nous étendons le principe d'isolation de domaines par les instructions. Ce principe est présenté par les auteurs

de RIMI [33] et consiste à dupliquer les instructions d'accès mémoire pour chaque domaine qui doit être isolé. De cette manière, les données contenues dans le code JIT ne peuvent être utilisées que par le code JIT lui-même et la "*shadow stack*" n'est accessible qu'en utilisant les instructions `load`/`store` qui lui sont attribuées. L'espace mémoire est séparé en plusieurs domaines qui sont présentés sur la Figure 1 avec leurs permissions associées. Le domaine de base (bleu) contient le code statique de la VM et de ses composants et utilise la pile pour ses appels et une zone dédiée du tas pour y stocker les objets utilisés par l'application. La zone JIT (vert) contient le code JIT, les données associées ainsi que les "*trampolines*", des routines directement émises en code machine qui sont utilisés par le code JIT. Les données de cette région sont placées séparément et dans une région qui est "*read-only*". Enfin, une autre pile d'appel est placée dans une zone à part et contient les adresses de retour des appels de fonctions.

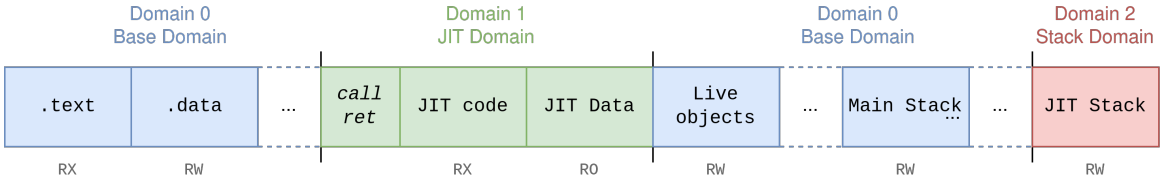


FIGURE 1 – Modèle mémoire de la séparation des domaines.

Pour supporter la séparation de ces domaines, plusieurs modifications sont implémentées sur le processeur CVA6 [3]. Tout d'abord, l'étage de décodage est étendu avec les nouvelles instructions d'accès mémoire depuis le code JIT (`l*1/s*1` pour les différentes variantes de taille des `loads`/`stores`), les instructions d'accès à la pile d'appel (`lst`/`sst`) ainsi que les instructions de changement de domaine (`chdom`/`retdom`). Chaque instruction se voit associer un domaine d'exécution et un domaine d'accès. L'attribution des domaines se fait par extension du module "*Physical Memory Protection (PMP)*" déjà présent dans le processeur et qui associe à différentes plages d'adresses leurs permissions associées. Ce module est étendu pour intégrer des informations de domaines, vérifiées une fois les permissions validées. L'accès aux données est validé dans ce module, intégré dans l'unité qui contrôle les accès mémoire. De même, les instructions de changement de domaine vérifient à la fois que leur domaine d'exécution correspond au domaine courant et le "*fetch*" d'instructions en résultant correspond bien au nouveau domaine. Enfin, les instructions de base du jeu d'instruction RISC-V se voient attribuer des accès au domaine de base ainsi qu'une vérification d'absence de changement de domaine.

Les modifications apportées au processeur sont minimales et correspondent à une

augmentation des ressources utilisées de moins de 0.5%. Nous instrumentons les binaires générés par Gigue avec cette solution pour déterminer l'impact sur la performance. Nous validons l'exécution des binaires dans le harnais de tests de Gigue en vérifiant le domaine courant et accédé de chacune des nouvelles instructions. Après instrumentation, et pour les mêmes classes d'application que celles définies dans le test d'interfaçage de Gigue, nous exécutons les binaires sur le processeur modifié. L'impact sur la performance est minimal sur l'intégralité des classes d'application, et s'inscrit à 1.5% en moyenne pour le nombre de cycles, et 0.95% pour le nombre de cycles par instruction.

Extension d'un Compilateur JIT sur RISC-V

Pour pouvoir proposer un cas d'usage réel sur la mise en place de solutions d'isolation légère, nous étendons le compilateur JIT de la VM Pharo sur le jeu d'instruction RISC-V. Ce port est motivé par l'absence d'un compilateur JIT complet sur ce jeu d'instruction et la validation de résultats préliminaires obtenus à travers les binaires générés par Gigue. La VM Pharo utilise un harnais de tests complet et un environnement de simulation qui permettent de faciliter le port de son compilateur JIT surde nouvelles architectures. Le langage Pharo, successeur de Smalltalk-80 utilise des "*images*" comme support d'exécution de ses applications, contenant à la fois le code source des principaux éléments du langage, des outils dédiés ainsi que les objets instanciés par la VM.

Le compilateur JIT de la VM Pharo, Cogit, est un compilateur non-optimisant qui recompile les successions de bytecode à la granularité d'une méthode. Il utilise une représentation intermédiaire à deux adresses, calquée sur le jeu d'instruction x86/x64 pour permettre sa traduction directe. Ce choix de la représentation intermédiaire a nécessité plusieurs adaptations dans le port du compilateur sur RISC-V. Le jeu d'instruction RISC-V fait plusieurs choix de simplification qui permettent de faciliter son implémentation dans un processeur. Parmi eux, l'existence d'un unique mode d'adressage de la mémoire, l'absence de codes de conditions ("*flag registers*") et de leurs instructions de comparaison associées, ou l'unique utilisation de valeurs immédiates. Pour faire coïncider le processus de compilation avec RISC-V, nous rajoutons une étape dans la compilation pour gérer l'utilisation des codes de conditions et traduire une suite de comparaison/branchement correctement. Nous rajoutons également des vérifications sur le traitement des valeurs immédiates utilisées et de leur correcte extension de signe.

Pour pouvoir valider le port de Cogit sur RISC-V, nous étendons l'environnement de

simulation propre à la VM Pharo [41], [42] en utilisant une nouvelle version du simulateur de processeur utilisé dans le harnais de tests, Unicorn [40]. Ce simulateur est utilisé en boîte noire pour valider les différents comportements du JIT. Plusieurs bugs ont été corrigés dans ce simulateur et rapportés dans ses dernières versions. Le harnais de tests a aussi été étendu avec des vérifications propres au port sur RISC-V et pour valider les changements dans la représentation intermédiaire ainsi que contrôler l'expansion du code généré pour combler les instructions manquantes sur RISC-V. Enfin, le débogueur de code machine implémenté dans l'infrastructure de tests est également étendu pour afficher plus clairement la traduction entre la représentation intermédiaire et le code machine généré. Nous étendons le simulateur pour supporter des instructions "*custom*" spécifiées par le développeur et validons leur exécution dans l'environnement de simulation.

La VM et son compilateur JIT sont finalement déployés sur une image QEMU [43] pour mesurer le gain en performance. Nous remarquons à l'aide de "*micro-benchmarks*" que le compilateur JIT permet une exécution de bytecode 40x plus rapide (en bytecode par seconde), ainsi que 63x plus d'appels de fonctions par seconde. Cependant, pour constater l'impact des patches nécessaires à la concordance entre la représentation intermédiaire et le code machine généré, nous constatons également une augmentation de 48% de la taille des méthodes JIT compilées par rapport à leur version x86/x64. Le port du compilateur JIT ouvre la porte à une expérimentation sur la pertinence d'instructions dédiées au compilateur dans un contexte de sécurité mais également d'accélération matérielle plus général.

Conclusion

À travers ces travaux, nous proposons, mettons en œuvre et évaluons un cadre de sécurité soutenu par le matériel pour protéger le code JIT. Il applique des politiques de sécurité à la région du code JIT, comme le prévoient les défenses de l'état de l'art autour des machines virtuelles, à un coût matériel et logiciel minimal. Pour évaluer l'impact de la solution et la comparer à d'autres, nous proposons également Gigue, un générateur de binaires instrumentables. Enfin, nous avons étendu le compilateur JIT du Pharo VM pour qu'il prenne en charge l'ISA RISC-V et définit les premières étapes d'un cadre de sécurité associé. Nous pensons que ces trois contributions, disponibles en open-source, fournissent un environnement réaliste pour de futures évaluations et extensions dans les domaines de la sécurité matérielle et de son application aux machines virtuelles langage.

Chapter 1

Introduction

1.1 Motivation

Over the last decade, the security of an increasing number of computing systems has been compromised. One of the most well-known and severe vulnerabilities was disclosed in 2014 as the "Heartbleed" bug [1]. It affects the widely-used OpenSSL cryptographic software library [2] through its implementation of the Transport Layer Security (TLS) protocol "*heartbeat*" extension (RFC6520 [44]). The use of unmonitored user input led the server to leak its secret keys and sensitive user information. Due to its usage in widespread open-source web servers like Apache and nginx, the bug was estimated to affect between 24 and 55% of popular HTTPS sites [45]. Using a different attack vector on the same target [46], Meltdown [47] and Spectre [48] exploited critical architectural vulnerabilities in Intel, AMD and ARM processors, therefore affecting billions of devices. These attacks managed to leak sensitive information (such as passwords or personal details) from the speculative execution of the processor, either in out-of-order execution or branch prediction. The security of computing systems is guaranteed by the implementation of software patches set up once the vulnerability is revealed and identified. To this end, OpenSSL was patched to mitigate the Heartbleed bug [1], and directives to protect against Meltdown attacks at kernel and OS levels since completely changing the hardware might not be an option. This leaves an open window for attackers where the bug is disclosed and users are vulnerable. In addition, even with updates patching the vulnerabilities, several platforms are not updated and live with the vulnerability.

To provide end users with guarantees on their application code security, several layers of primitives are set up at the hardware level, Operating System (OS) level, virtualization level, or through dedicated run-time environments [2]. Language Virtual Machines

1. `https://git.openssl.org/gitweb/?p=openssl.git;h=96db9`
2. We use the word "run-time" as an adjective to qualify the dynamic nature of an object (*e.g. "run-time solution"*), "runtime" as a noun to define the execution environment of a language (*e.g. "the C# runtime"*), and "run time" to qualify the moment of the execution (*e.g. "at run time"*).

(VMs) are the execution environment of high-level managed programming languages. They abstract the compilation process and memory management to the developer and offer portability of application code. Smalltalk, Java, Python, or JavaScript all use a VM to execute and manage their code. They are complex engines composed (at least) of a source code compiler, an interpreter, a memory manager, and an optional set of increasingly optimizing run-time compilers that recompile frequently used code at run time. Their volatility and portability make them easy to deploy on several devices while guaranteeing the execution of application code. For example, JavaScript VMs are deployed in all web browsers and execute the embedded JavaScript code sent by the server on the client computer. To this end, most web browsers embed either the Google V8 engine, Apple JavaScriptCore, or Mozilla Firefox SpiderMonkey. Therefore, billions of devices deploy a JavaScript VM to display the latest ECMAScript standard [49], the reference to guarantee the inter-operability of the modern web. Other widely deployed examples include Android applications that are Java applications compiled down to the intermediate representation of the Java Virtual Machine (JVM), Java bytecode. This bytecode is then translated into an executable to run on the Dalvik Virtual Machine (DVM) or its successor, Android RunTime (ART). The executable is distributed to billions of end users and their devices.

The wide deployment and adoption of VMs coupled with their ability to handle memory and output machine code at run time makes them interesting targets for attackers that aim at end users. Traditional code injection attacks install attacker-supplied input in executable memory before redirecting the flow of control [3] to it. They are amplified in the context of VMs as the Just-In-Time (JIT) compiler(s) use a memory zone to deploy optimized and executable machine code. This zone is modified by an attacker through the injection of a malicious payload [4], or the reuse of genuine code that contains predictable elements that are then composed to define a payload [5], [6]. Other attack vectors through other components such as the interpreter [9] or garbage collector [50] can be exploited to leak sensitive information, corrupt the state of the VM, or, in the worst case, lead to arbitrary code execution on the victim's computer. Through the widespread distribution of VMs presented earlier, an attacker could launch their attack by targeting an end user with a malicious webpage or Android application to remotely execute arbitrary code. A recent example of a major vulnerability involving a VM is *Log4Shell* [51], which affected

3. We use the appellation "control flow" or "flow of control" to name the order in which program elements are executed (statements, functions, or instructions). We use the adjective "control-flow" accordingly (*e.g. "control-flow integrity"*).

the popular *Log4J* Java logging framework. The unmonitored user input was used to inject Java objects from an arbitrary URL to target malicious endpoints, therefore enabling arbitrary Java code execution on the server.

1.2 Formulation of the Research Question

The traditional answer to disclosed vulnerabilities is to patch the issue in the affected component and provide updates for the end users. Admitting the possible weaknesses of parts of the system and building defense solutions with it in mind mitigate the overall impact of a later attack. Preemptive actions and primitives that enforce security properties are highly desirable in a context that handles untrusted user input. Software-only proposals such as Control-Flow Integrity (CFI) [52], Information Flow Control (IFC) [53], or Data Flow Integrity (DFI) [54] offer strong security guarantees to the user. However, along with the guarantees come a heavy cost in performance and code size, preventing the actual usage of such solutions in performance-critical systems. While those solutions apply to the static part of the VM, dynamically generated code and data are too expensive to instrument [18] and result in degraded versions of the solution [19]. Common security measures such as guaranteeing that memory is never executable and writable at the same time are not trivial to enforce on dynamically generated JIT code [10], [19] and often result in a deterring cost in performance.

The integration of hardware-accelerated security features into vendor processors offers a tangible performance advantage over their software counterparts. This makes them attractive for performance-critical components and ensures robust isolation when deployed around and in VMs [9], [10]. However, challenges arise when dealing with closed-source vendor solutions such as Intel SGX [21, Vol.3 Ch.34-39], or ARM TrustZone [55]. Responding to a disclosed vulnerability targeting the mechanism becomes difficult due to the patch cycle aligning with the next generation of processors from the vendor. Many of these extensions were deprecated by the manufacturer itself (*i.e.* Intel MPX, Intel SGX, *etc.*). The necessity for open and flexible solutions becomes evident, especially in light of vulnerabilities identified in their counterparts. Dedicated instructions offer an alternative approach to existing solutions [22], [32], or provide new fine-grain isolation techniques [31], [33]. We raise questions about their application or design, particularly concerning dynamically generated JIT code.

We present the thesis statement as the following:

"Dedicating custom instructions supported by the underlying hardware processor to the machine code generated by the JIT compiler of a VM can enforce strong isolation primitives around this performance-critical component."

Derived from this statement, we define three research questions that tackle the different research axes of this thesis:

`RQ1:` What metrics define hardware solutions in the context of JIT code, and how can we compare solutions without complete access to a VM?

`RQ2:` What defines an interesting custom instruction for the JIT code, and how can the JIT code be isolated through the instructions it generates?

`RQ3:` What challenges arise when adding custom instruction to the JIT compiler, and how can it be tested without complete access to the hardware implementing the solution?

1.3 Contribution

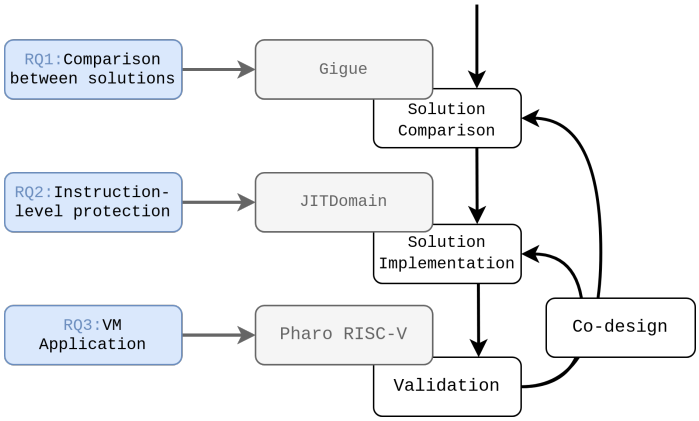
In this thesis, we propose a security approach that shifts away from relying solely on software-defined security primitives. Instead, we advocate for a strategy based on leveraging dedicated hardware-assisted security features. Furthermore, rather than utilizing off-the-shelf components found in commercial processors, our proposal involves extending open-source processors through a defined methodology to create and implement a comprehensive solution framework. Our method taps into the extensive instruction set extension space of the RISC-V Instruction Set Architecture (ISA) [20], [56], enabling versatile prototyping within existing processors. The proposed multi-level approach emphasizes early validation of prototyped solutions at different stages of the technology stack, ranging from processor support to practical usage in VMs. To assist in decision-making, we provide tools for estimating the potential impact of a solution on the JIT compiler performance before deploying the solution in a production VM. To this end, we reconcile low-level hardware extensions to the high-level JIT compiler through the instrumented generated machine code. The main contributions of this thesis are the following.

A synthetic JIT code workload generator. We introduce Gigue, a synthetic random workload generator designed to produce executables resembling JIT code regions. The generated binaries are self-contained and can be deployed bare-metal on the processor, either directly or through their cycle-accurate simulator. We present how Gigue can be utilized to generate a set of binaries and augment them by shifting their input parameters. We run the resulting binaries on two fully-featured RISC-V cores, CVA6 [3] and Rocket [39]. In addition, we present how we integrate custom instructions in the Gigue binaries through hooks available at different instruction generation levels. This extension is also present in the testing environment Gigue provides to validate to test the correct execution of binaries, even when containing custom instructions. We also provide utilities to run, collect, and display results, as well as patch the compilation toolchain or generate minimal binary examples.

An instruction-level domain isolation implementation. We present JITDomain, an instruction-level domain isolation framework derived from embedded systems [33] and applied in the context of JIT compilation. It takes advantage of the locality of the JIT code region and its need for strict and performant security guarantees. To this end, we add duplicated memory access instructions, routines to change domain, and shadow stack instructions. The JIT code is allocated within a hardware-defined domain, enforcing several aspects: *data separation* as only JIT code is permitted to access JIT data present in the JIT code region; *call stack separation* as the JIT call stack is placed into a domain accessible exclusively through the prologues and epilogues of JIT-compiled methods; *system-call filtering* from the JIT code, where the execution is stopped as soon as a system call instruction is decoded from the domain, either from the expected or hidden (after a disruption) control flow. We propose an implementation of the design in the CVA6 RISC-V core [3] and present an evaluation of the solution on Gigue binaries that implement the solution. The resulting overhead in terms of FPGA resource utilization boils down to a 0.5% area overhead, and the performance overhead is measured at less than 2%.

A JIT compiler port to the RISC-V ISA. We highlight the challenges of porting a JIT compiler to the RISC-V ISA by presenting the port of Pharo VM's JIT compiler Cogit to the RISC-V ISA. We present the main differences between the RISC-V ISA and more traditional x86/64 or ARMv7/v8 instruction sets. The impact of these differences is presented through the lens of the Pharo VM JIT compiler Intermediate Representation (IR) and on the testing and tooling ecosystem. We extended the Pharo VM development

tools to accommodate the available RISC-V environment. Among them, is Unicorn [40], the CPU emulator used to test JIT code from within the Pharo environment, and the machine code debugger. We developed a version compliant with the Pharo VM's test harness and managed to run a minimal image on a QEMU RISC-V Fedora image getting 40x speedup over the base interpreter in the number of executed bytecodes and 63x calls per second. To add JIT code custom instructions, we connect emulator hooks to handle custom instructions, allowing for prototyping and validation of custom instructions and their usage from within the VM development environment.



The contributions are linked to the research question they aim to answer in the above figure. Their goal and articulation range from the creation of a platform for meaningful comparison of solutions, to the implementation of a solution in hardware, along with a port of the Pharo VM and a first implementation in its simulation environment. The wider goal of this thesis is to provide tools to co-design hardware isolation for VMs.

1.4 Organization

The remainder of this thesis is outlined as follows. In Chapter 2, we conduct a comprehensive examination of the background and state-of-the-art attacks targeting language VMs, along with the defensive measures implemented in response. We then identify a set of practical components and their associated requirements for VMs to consider adopting. Chapter 3 delves into the RISC-V instruction set and the corresponding extensions, viewed through the lens of the identified VM requirements. The chapter sheds light on the solutions derived from this exploration. Next, in Chapter 4, we introduce Gigue, our workload generator. Gigue serves as a foundational tool for comparing security solutions

in the context of just-in-time-compiled code. Chapter 5 details our efforts in designing and implementing JITDomain, an instruction-level domain isolation framework that exploits just-in-time-compiled code locality for isolation. Moving forward, Chapter 6 addresses the challenges associated with extending the Pharo VM JIT compiler to the RISC-V ISA and how it opens up further experimentation for VM security prototyping. Finally, in Chapter 7, we discuss the future directions of this work and conclude the thesis.

Part I

Background & Related Works

Chapter 2

Related Work: Language Virtual Machines Security

In recent years, security has become a crucial issue, and the technical complexity associated with it continues to grow. Thus, security attacks and defenses follow the usual cat and mouse game of finding a new attack and designing a security mechanism to protect against it. This has led to complex, heterogeneous attacks, i.e. those operating at several levels, to which fine-grained defense mechanisms respond.

This concern also applies to language Virtual Machines (referred to as VMs, for which an overview is provided in Section 2.1). Attackers attempt to compromise VMs and, through it, access the victim host computer. Section 2.2 covers various attack schemes. Defense solutions, as presented in Section 2.3, must strike a balance between security guarantees and the performance overhead they incur. As the global summary presented in Section 2.4, fine-grain isolation of known critical VM components is crucial for its protection. Performance-critical components benefit from hardware-enforced guarantees due to their lower impact on performance and the higher level of security they provide.

2.1 Background: Language Virtual Machines

Glossary: In the book "Virtual Machines: Versatile Platforms for Systems and Processes" [57], Jim Smith and Ravi Nair introduce a taxonomy of the term Virtual Machine and its different meanings. They split the VM characterization into two main categories, as presented in Figure 2.1: process VMs and system VMs. Process VMs support an Application Binary Interface (ABI), user instructions, and system calls. Process VMs using the same guest/host Instruction Set Architecture (ISA) come as multiprogrammed systems (as included in most of today's systems) or dynamic binary optimizers that optimize guest instructions. Process VMs with different guest/host ISA come as dynamic translators that convert guest instructions to the host ISA, or High-Level Languages (HLL)

VMs that perform an abstraction of the guest at a higher-level process interface. System VMs support a complete ISA, with both user and system instructions. Their main goal is to provide replicated and isolated system environments of a guest ISA on top of another (or the same) host ISA.

This chapter addresses HLL VMs, which will be called Language VMs, VMs, or execution engines transparently, with a specific focus on security, from both attacks and countermeasures points of view.

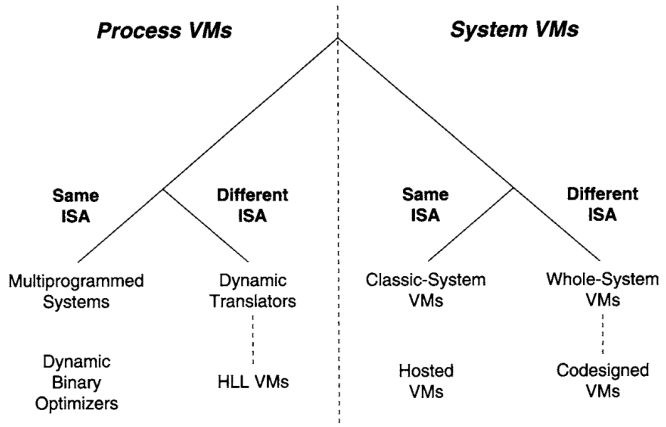


Figure 2.1 – A Taxonomy of Virtual Machines, as presented by J. Smith and R. Nair [57].

Language virtual machines have been a major milestone in language implementation through the two main contenders Smalltalk [58] and Java [59]. They simplified the process of ensuring the portability of applications for developers by providing a portable execution environment. As Java advertised in 1995, "write once, run anywhere" (as long as the Java VM is available for the corresponding OS and architecture). Smalltalk and Self [60] have brought to life many of the concepts that are still used today in the virtual machines of recent languages. Among them, careful language optimizations [35], [61], [62] and adaptation to new architectures [63], [64]. Today, virtual machines are widely deployed. In cloud environments, as the run-time environment for supported languages; as embedded support in web browsers to execute the JavaScript client-side part of web pages. However, their propagation, widespread usage, and implementation complexity make them a target of choice for attackers.

Before analyzing these attack scenarios, a deep understanding of the insights of virtual machines is mandatory. These components, an overview of which is depicted in Figure 2.2, will be further detailed in this section, along with corresponding annotations in the

figure.

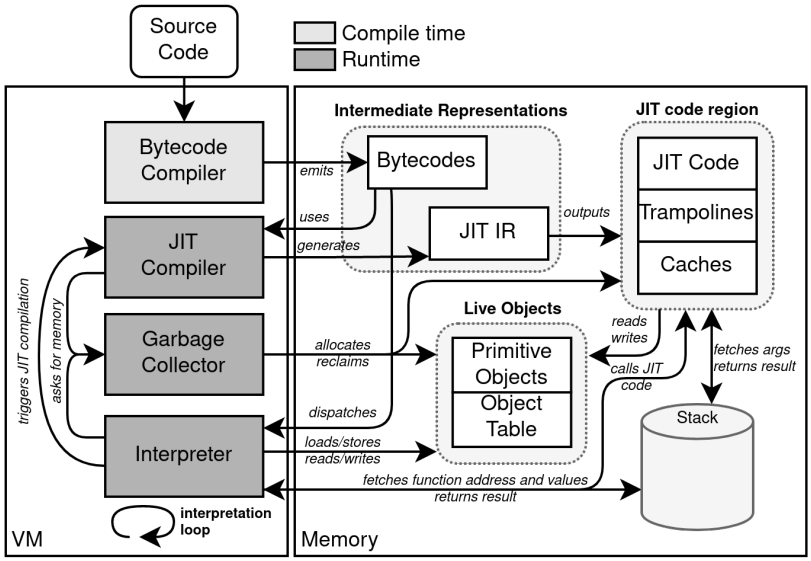


Figure 2.2 – Virtual machine overview (derived from [9]).

2.1.1 Interpretation

To start the execution of a managed language, the source language is *interpreted*. This step starts with a *parser* that constructs the Abstract Syntax Tree (AST) representation of the input source language. The AST is then compiled by a *bytecode compiler* into a VM-specific intermediate representation (IR) referred to as bytecode. The bytecode is consumed by an interpreter, in a cycle of fetching the bytecode, analyzing it, executing the corresponding action, then fetching the next bytecode. As most of the interpreter activity `decode/dispatch` is spent in the dispatch loop, the base process tied in both the direct and indirect branch predicts that impact performance [65]. "Threaded interpretation" [66] promotes a `goto` implementation rather than the base `switch` implementation [67]. This allows the dispatch step to occur by jumping directly to the next correct execution unit at the end of the previous execution unit, reducing the time spent in the loop. However, with more recent processors and more precise branch predictors, the indirect branching prediction now presents less of an impact on performance [68] and any optimization on the interpretation process usually fall short in the presence of a Just-in-time (JIT) compiler.

2.1.2 Just-in-Time Compilation

JIT compilation refers to the recompilation at run time of a succession of bytecode detected as of interest as being frequently called. Hence, in *mixed mode Just-in-time (JIT) compiler architecture*, the interpreter, baseline of the execution is followed by a sequence of successively higher optimizing JIT compilers. The qualifications of "succession of bytecode" and "frequently" from the previous sentence are fine-tuned and VM-dependent. The granularity depends on the VM: either it is an execution trace [69], [70], a method [59], [71], or a basic-block [72]. Once a "hot path" is detected, the selected JIT compiler (which is selected based on its degree of optimization) recompiles the succession of bytecode into a compiler-specific Intermediate Representation (IR). The need to introduce this new IR comes from its different goals. Whereas the bytecode is designed to be executed by the interpreter, the JIT IR is designed to support optimization down to machine code.

The two main challenges faced by JIT compilation are: (1) ensuring `calls/returns` between the interpreter and the JIT code region transparently, and (2), in the case of dynamically typed languages, type-inferring efficiently to avoid a costly method lookup.

Control-flow dispatch: The call stack of the JIT code region needs to interact with the native stack of the VM as well as the rest of the host architecture. To help redirect control flow, switch between the native and JIT call stack along the defined calling convention, as well as perform other often-used routines, the JIT compiler installs machine code stubs called *trampolines* in the JIT code region. Direct call to these trampolines avoids the use of duplicated routines throughout elements of the JIT code region.

Type inference: Dynamically-typed object-oriented languages are prone to slowdown at *method-lookup* time. Identifying the correct class from which to extract a named method for application to the final instance is a costly process. It is usually sped up through a collection of optimizations. *Inline caches* specialize the type by generating machine code for the most used receiver type and verifying it with a type guard. This type specialization can be propagated aggressively through the whole trace [73] of a tracing compiler. It can also be extended to *Polymorphic Inline Caches (PICs)* [35], a machine code jump table that redirects to the correct polymorphic method in the JIT code region based on the receiver type. From the study of the usage of PICs, often used receiver types are inferred to better direct caching on further executions in an optimization called *type feedback* [74]. *Hidden classes* (as named in Google Chrome JavaScript V8 engine) or *maps* [61] are introduced in Self, a dialect of Smalltalk, and help to efficiently manipulate dynamic objects by creating a shared iterative underlying structure that stores the layout

of live objects. Splitting the layout from the live objects and having them reference it makes any optimization of the layout beneficial to all live objects referencing it and reifies the JIT compilation of shared structures.

2.1.3 Additional components

In addition to the components responsible for the compilation and execution of the source language (that is the main focus of this thesis), VMs need two parts to fully operate: (1) a component that transparently manages memory for the applications it executes and (2) a way to operate with the lower-level components of the execution stack: the Operating System (OS) and shared libraries. VMs automatically grant and retrieve memory to objects in the heap. The component responsible for memory allocation and retrieval is called a *garbage collector* [75]. This component is a research area by itself, both in the algorithms it uses, the performance it provides, and the treatment of garbage collector-related bugs (which are often very complex). To fully integrate into modern computer architecture, VMs also need to interact with external elements handled by the OS through shared libraries or other routines. *Foreign function interfaces (FFI)* connect the VM to its surroundings and allow it to perform OS and architecture-dependent tasks such as `syscall` accesses, graphic libraries, or user-defined bridges to routines in other languages.

2.2 Virtual Machine Attacks

Attacks against VMs are based on known attacks against statically compiled code, extended in the dynamic context of VMs, and taking advantage of the low-level operations the JIT compiler has to perform. Attacks are divided into three main categories: **code-injection attacks**, **code-reuse attacks**, and **data-only attacks**. These categories are mostly presented in chronological order and highlight the vulnerability of many parts of the VM. In the rest of this chapter, the following terminology is used to qualify attacks and the elements surrounding them:

— *Vulnerability:* A weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source. Common memory vulnerabilities are: *buffer overflow*, from lack of array bound checking, allowing an attacker to spill values on the stack; *use-after-free*, from dangling pointers, referencing uncontrolled memory; or *type confusion*, from object metadata confusion, giving access to memory referenced for object attributes.

— *Exploit:* A program, piece of code or even some data written by a hacker or malware writer that is designed to take advantage of a bug or vulnerability in an application or operating system.

— *Payload:* Objective of what a virus, worm or trojan is designed to do on a victim's computer, either delivered via a stager or directly placed on the victim's computer (called stageless, or stage-0).

— *Stager:* An executable used in multistage cyberattacks to establish a connection to the command-and-control (C&C) server, deliver and then run larger malicious modules.

— *Shellcode:* A type of payload that spawns a `shell` for the attacker, by running `"/bin/sh"`, often hardcoded in assembly.

2.2.1 Context and Threat Model

Context: In their work, Zhang et al. [19] illustrate the process that the Firefox Gecko engine follows to execute and display internet resources. HTML, CSS, and multimedia inputs are first consolidated into a *Document Object Model (DOM)*, which is utilized by the *layout engine* and *rendering engine* to present web pages to users. The DOM interface also serves as a communication channel for the JavaScript VM (SpiderMonkey in Firefox) to interact with page elements. Scripts undergo an initial interpretation and are then

recompiled by the JIT compiler to generate platform-specific native instructions. These instructions are placed in the JIT code region, where they may be optimized as described earlier. The garbage collector manages memory allocation and reclamation for necessary objects. Major web browsers, such as Firefox, Chrome, and Safari, embed production JavaScript VMs (like SpiderMonkey and V8). These VMs continuously evolve to meet the latest standards and user demands for high performance. However, each year, new memory vulnerabilities are discovered in production VMs, often related to memory errors or data validation, which can potentially lead to arbitrary code execution on the host computer.

Threat Model: In line with the scenario and expectations surrounding modern VMs, the fundamental threat model for VM attacks assumes that the attacker can supply arbitrary input data (source code) to the VM. This input is expected to be compiled, executed, and possibly recompiled by the VM without raising any suspicion. The system incorporates static code defenses, including strong $W \oplus X$ policies, Data Execution Prevention (DEP), and Address Space Layout Randomization (ASLR). These defenses have been designed to counter overflow attacks, either they target the stack or the heap. In these attacks, an adversary typically injects malicious code into executable memory and then manipulates the control flow to execute it. In the remainder of this document, the threat model is refined to better represent the capability of the new attacks. As new vulnerabilities are found each year in VMs, it is expected that an attacker knows the existence of a memory vulnerability.

Orthogonal works: Side-channel attacks exploit information from hidden auxiliary channels, which can be either physical through time or energy leaks [47], [48], or software-induced using memory locality properties [76]. JIT compilers, susceptible to introduce timing side-channel vulnerabilities [77] in the generated machine code, have been addressed in the literature [78], [79]. Garbage collectors, responsible for handling sensitive objects, are also vulnerable to side-channel attacks [50], [80], potentially revealing information on disposed objects. We argue that these side-channel attacks, though orthogonal to our work, should be considered in the design of a VM protection solution.

2.2.2 JIT Code Injection Attacks

A primer on code injection attacks: The Phrack article "Smashing the stack for fun and profit" by user Aleph One [81] presented in 1996 how to use a buffer overflow exploit to redirect the control flow to the stack where a shellcode is executed. This kind

of attack, targeting the stack or heap, is called a code injection attack. It leads to the execution of arbitrary code through a *shellcode* (a piece of code that spawns a `shell`) for the attacker and a totally compromised system. In modern attacks, the shellcode is staged and split into several parts with dedicated goals such as: allocating memory with all permissions (*RWX*), downloading a payload (larger piece of code to execute), or triggering a vulnerability. The *"stack pivot"* [82], [83], is a stub of code that, when executed, switches the stack from the genuine application program to the injected malicious code (*e.g.* a fake "stack" in the heap).

Threat Model Refinement: The defenses set up by the base threat model presented in the previous section should mitigate classic code injection attacks on the statically compiled code of the VM.

Application to VMs: JIT compilers initially wrote all run-time-generated code onto memory pages that were simultaneously writable and executable throughout the execution. Since the code must be written at run time to be executed, it must use memory that is both writable *and* executable. This lets an attacker perform *code-injection attacks* such as the one presented by Gong et al. [84] where he injects a malicious payload into the JIT memory to gain arbitrary code execution in the Chrome browser.

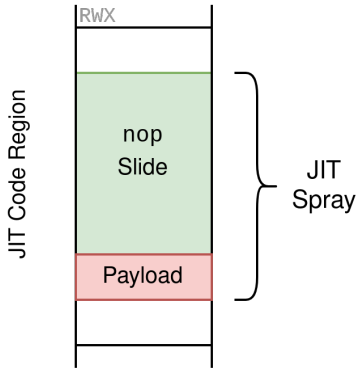
A new category of code injection attacks called *JIT spraying* attacks is initially presented by Dion Blazakis [85]. This type of attack includes large constants (or `XOR` chains of large constants) that hide a sequence of native machine code instructions when shifted by a given number of bytes. Later, a disruption of the control flow reveals the hidden instructions composing the exploit.

Figure 2.3 is an example of such an attack and can be split into three parts: on the left, extracted from the original exploit presentation, the ActionScript statement that `XOR`s three large constants; on the right, the corresponding machine code instructions when compiled by the ActionScript JIT compiler. As the generated code using the immediate values is stored in executable memory, these values can be interpreted as other instructions by shifting the execution point.

The sequence is composed of a `NOP` succession and `cmp al` instructions, semantic `NOP`s that consume the original `xor eax` opcodes (`0x35` bytes) as operands. This chain can be extended at will and means that as long as execution begins at any of the unintended instruction boundaries (*i.e.* anywhere but the middle of the `CMP` instruction), the `NOP` chain executes. As long as the most significant byte of each immediate is `0x3c`, the `XOR` opcode will be masked into a `CMP`. The shellcode can then be placed at the end, replacing

```
var x = 0x3c909090
      ^ 0x3c909090
      ^ 0x3c909090
      ...

for (i=0; i != 10000; i++){
      function(x)
}
```




```
# Format - addr: (raw instr) disasm instr

# Instruction sequence starting at 0x100
100: (0xb89090903c)   mov eax, 0x3c909090
105: (0x359090903c)   xor eax, 0x3c909090
10a: (0x359090903c)   xor eax, 0x3c909090
...

# Instruction sequence starting at 0x101
100: (0xb8)          # Skipped
101: (0x90)            nop
102: (0x90)            nop
103: (0x90)            nop
104: (0x3c35)          cmp al, 0x35
106: (0x90)            nop
107: (0x90)            nop
108: (0x90)            nop
10a: (0x3c35)          cmp al, 0x35
...
```

Figure 2.3 – JIT spraying example.

the `0x90` chains with instructions up to 3 bytes in length. Filling memory with a long `NOP` chain defeats the idea of randomization, as executing the `NOP` chain at any point brings to the shellcode (as shown in the bottom left part of Figure 2.3). An important aspect to consider is that the vulnerability used to divert the control flow can originate from both the VM and the JIT code itself.

The attack has then been extended by Alexey Sintsov, who demonstrated an ActionScript JIT spray shellcode for x86 [4] and for other JITs such as the JavaScriptCore Baseline JIT (embedded in the Safari web browser). Rohlf et al. [86] extend the method against Mozilla JaegerMonkey and TraceMonkey JavaScript engines (embedded in the Mozilla Firefox web engine) on the x86 architecture. Gawlik et al. [87] extend the idea to the Mozilla Firefox ASM.JS compiler, a module that uses a subset of JavaScript and is compiled *ahead-of-time*. They manage to perform a new version of the JIT spraying attack, forcing large constants to be sprayed in the JIT region.

As this type of attack was first performed on the x86 architecture, due to its lack of instruction alignment requirement, the question of feasibility was raised for other architectures that have stronger requirements, both in terms of alignment and calling conventions. Lian et al. [7], [8] presented JIT attacks on ARM and how JIT spraying applies. By forc-

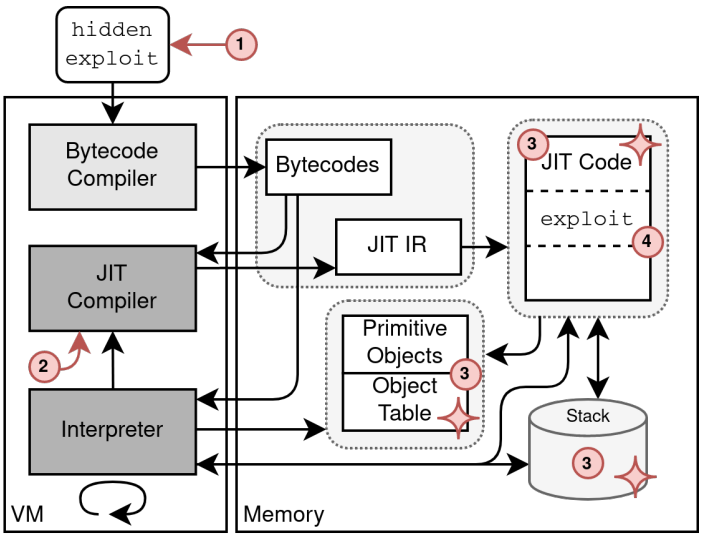


Figure 2.4 – Code injection attacks summary.

ing the emission of 32-bit ARM `AND` instructions, they have control of 20 bits over each and can make them interpreted as two 16-bit Thumb-2 instructions (ARM compressed format). In this way, the first instruction is used to perform useful operations for the attacker, and the second as an unconditional `PC`-relative forward branch to the next unintended instruction.

A summary of code injection attacks against JIT compilers is shown in Figure 2.4. An attacker injects the payload ① directly through the inputs of the VM as large constants in plain source code. Then they manage to trigger the JIT compilation ② by making the source code used enough (loops are usually sufficient to trigger the JIT compilation). Finally, a memory vulnerability ③ in live objects, the native stack, or the JIT code itself is used to redirect the control flow ④ to the exploit.

Summary: JIT spraying attacks blurred the lines between code injection and code reuse attacks and provided a new ground for attackers. While it is extremely powerful on the x86 architecture and resulted in a clear mapping of attacker-supplied input to executable memory, the attack becomes more convoluted (but not impossible) on other architectures that enforce stronger requirements on executable code.

2.2.3 Code Reuse Attacks

A primer on code reuse attacks: In a 2007 article, Hovav Shacham defines the concept of *Return-Oriented Programming (ROP)* [88]. The objective is to identify gad-

gets, units of machine code that hold a particular function (*i.e.* load, store) and end with a `ret` instruction. Combining them in the correct order using their successive return addresses in a ROP chain allows the attacker to execute arbitrary code when redirecting the control flow to the chain. To identify gadgets, code-reuse attacks traverse and dynamically disassemble code pages, then construct the ROP chain at run time. The author presents the *Galileo* algorithm to statically analyze executables and libraries searching for gadgets. The most common ROP payload chains together gadgets that change the stack pointer to an attacker-controlled payload, allocate `RWX` memory, copy a larger shellcode payload into it, and execute it. To locate gadgets, attackers make use of the deterministic nature of the heap allocator when handling carefully crafted memory allocation/retrieval operations to defeat ASLR. This technique named *Heap Feng-Shui* comes from the 2007 article "Heap Feng-Shui in JavaScript" [89] by Alexander Sotirov.

Threat Model Refinement: On top of the existing static code protections, defenses now involve strong defenses against JIT spraying on dynamic code: diversification of the output code through NOP insertion or code randomization and enforcement of the `W^X` policy on the JIT code region (see Section 2.3.2). The scenario remains the same, and the attacker is expected to know the existence of a memory vulnerability.

Application to VMs: ROP attacks are amplified and especially interesting against JIT compilers. While library modules may contain easier-to-find gadgets, JIT page allocations are more predictable and under the influence of untrusted inputs. If the attacker (1) knows how to trigger JIT compilation and (2) has an idea of the resulting machine code, then he can generate arbitrary gadgets in machine code memory. These assumptions are plausible as (1) loops are often a sufficient way to generate compute load and trigger the JIT compiler, and (2) similarly to Heap Feng-Shui [89], JIT Feng-Shui [86] uses language abstractions to compel the JIT compiler to produce predictable code chunks and therefore useful gadgets (called *gaJITs* by the authors). Even though the ROP technique has to go through ASLR, an attacker can still force the engine into allocating many copies of the native code throughout memory. Additionally, all coarse-grained ASLR solutions (*i.e.* per module) are vulnerable to *memory disclosure* attacks where even a single run-time address is enough to disclose its page and relations with others. The two following attacks are presented in Figure 2.5.

Snow et al. have added JIT compilation in the ROP process to generate the payload with JIT-ROP [5]. For precision, JIT-ROP is not directly related to the JIT code region. It describes a technique to repeatedly locate, read, and disassemble static code and then

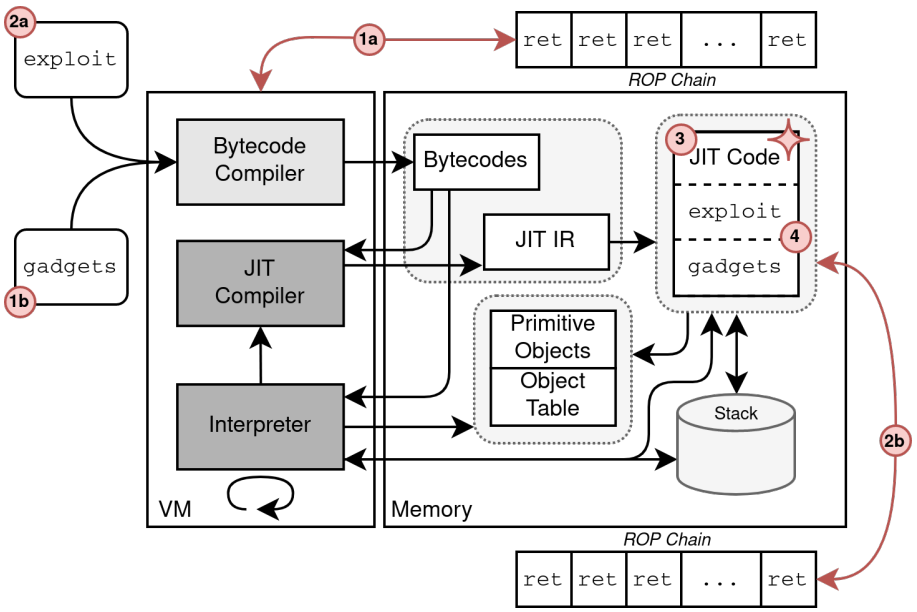


Figure 2.5 – Example of two ROP attacks against VMs [5], [6].

uses a code-reuse payload built with the JIT compiler. The exploit requires the attacker to use a memory disclosure to reveal a code pointer. Once the initial byte is disclosed and provided, code page harvesting begins (1a), detecting API to system or library calls, and discovering gadgets. The target program is specified in a DSL that compiles down to the ROP chain with discovered gadgets and APIs. This program is JIT-compiled into memory, along with an exploit buffer (2a) that transfers the control flow to the ROP chain. A call to the JITted function triggers the execution of the exploit buffer ③ and the payload ④. This idea consists of the usual ROP attack against static disassembled code, but helps defeat fine-grained ASLR that would be applied to the static binaries and would provide a way to trigger the payload through the use of the JIT compiler.

Another version of the attack, focused on the dynamically-generated code only, is presented by Athanasakis et al. [6]. They show a ROP attack only using gadgets from the generated JIT code. They expect the static system to be hardened against code-reuse attacks, which means the binary and libraries are compiled with G-Free [90], a compiler that produces binaries without gadgets. The authors use 2-byte constants to generate gadgets in memory, similar to JIT spraying. They attack IonMonkey (in Mozilla Firefox) and ChakraCore (in Microsoft Internet Explorer) and overcome their defenses. As presented in Figure 2.5, the needed gadgets are injected through the input source code (1b) alongside a memory vulnerability. The JIT code is then disassembled to construct a

ROP chain (2b). The control flow is passed to the ROP chain using the vulnerability in the injected memory ③ and the exploit is launched ④. In this way, the authors present a complete exploit that creates the tools it needs in the JIT code region to perform the complete execution.

The main goal is to use the system call `mprotect` in the case of Firefox on 32-bit Linux and `VirtualProtect` for Internet Explorer on 64-bit Windows. Both system calls modify the permissions of given memory pages. As an example, the gadgets required to perform both system calls are presented in Figure 2.6. For `mprotect` (on the left), the first two gadgets store the page address and region length in `ebx` and `ecx`. The next two gadgets zero out `eax` and copy `0x7d` (system call number for `mprotect`) to it (`al` corresponds to the lowest byte of the `eax` register). The two following gadgets zero out `edx` and copy `0x7` (code for Read/Write/Execute permissions) to it. Finally, the last gadget issues the system call. For `VirtualProtect` (on the right), the first four gadgets are the arguments for the system call: the page address (`%r8`), region size (`%r9`), protection code (`%rcx`), and output pointer to hold the previous protection value (`%rdx`). The last gadget is here to handle *stack pivoting*.

```
1 pop %ebx          (ret) ; page address
2 pop %ecx          (ret) ; region size
3 xor %eax, %eax    (ret) ; zero eax
4 mov 0x7d, %al     (ret) ; mprotect nb
5 xor %edx, %edx    (ret) ; zero edx
6 mov 0x7, %dl      (ret) ; RWX code
7 int 0x80          (ret) ; syscall
8 ; Linux 32-bit
```

```
1 pop %r8    (ret)  ; page address
2 pop %r9    (ret)  ; region size
3 pop %rcx   (ret)  ; protection
4 pop %rdx   (ret)  ; output
5 pop %rax   (ret)  ; stack adjust
6
7
8 ; Windows 64-bit
```

Figure 2.6 – Succession of gadgets to call `mprotect` (left) and `VirtualProtect` (right).

Summary: As in more classical attacks, ROP attacks and their variants (ROP without returns, JOP, SROP or BROP [91]–[94] etc.) remain powerful attacks up to this date. They can be found in all architectures, even those that do not rely on the stack to `call/return` such as ARM [7] or RISC-V [95], [96]. They require expensive defense mechanisms to be properly mitigated, especially in dynamically generated code.

2.2.4 Data-Only Attacks

A primer on data-only attacks: Other attacks focus on data only, or as Chen et al. [97] called them, *non-control-data attacks*. They come as an opposition to *control-data*

attacks, where the objective of the attacker is to alter the program's control data (*e.g.* return addresses and function pointers) to execute malicious code with the privilege of the victim process. *Non-control-data attacks* focus on security-critical data that applications use and manipulate, such as configuration data, user input, user identity data, or decision-making data. Hu et al. [98] extend the concept by creating *data-oriented gadgets* and unlocking *data-oriented programming (DOP)*, an attack technique that only uses data values for malicious purposes, maintaining complete integrity of the control flow.

Threat Model Refinement: On top of the existing static and dynamic code protections, defenses now involve strong defenses against JIT code reuse: strong CFI is enforced on both static and dynamic code. The scenario remains the same, and the attacker is expected to know the existence of a memory vulnerability.

Application to VMs: Data-only attacks against VMs have been designed to focus on the input of the VM internals. They take place at the interpretation stage through bytecode [9] or at the JIT compilation stage, focusing on the JIT intermediate representation (IR) [10] or compilation internals [99] and their execution process is presented in Figure 2.7. Without resulting in code injection or reuse, injecting malicious data to trick the JIT compiler or interpreter into generating and/or executing the payload itself is a way to compromise it.

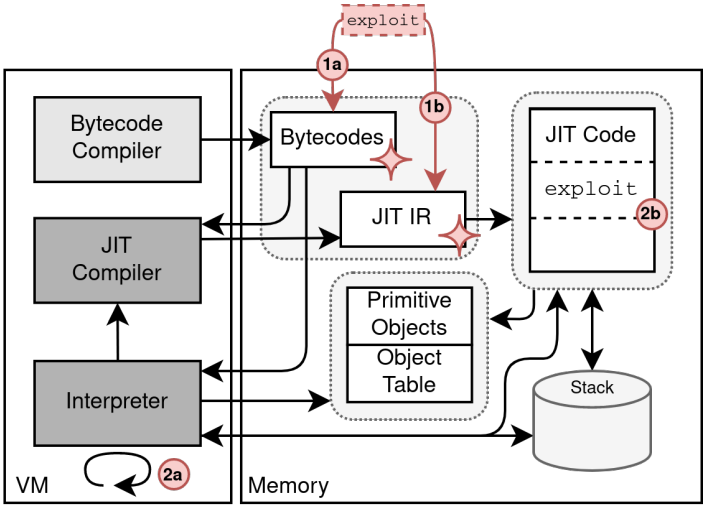


Figure 2.7 – Data-only attacks example [9], [10].

The JIT compilation process is vulnerable as well, through both its inputs and internals. Frassetto et al. [10] present a Data-Only JIT Attack (DOJITA) that targets JIT IR. They perform the attack against the ChakraCore VM according to the following process.

The attacker exploits a memory-corruption vulnerability to read and write arbitrary data memory. They identify a "hot" function in the input program. Then, they inject crafted C++ objects, the payload, into the IR (1b). The JIT compiler spreads the payload to the JIT code as its input comes from trusted bytecode. The generated code now contains a malicious payload that will be executed when the victim "hot" function is called (2b). On the other hand, the R&D team Theori [99] presents an attack that targets ChakraCore too, bypassing the JIT Control Flow Graph (CFG). It targets a temporary buffer used during compilation that is first readable and writable, and then marked as readable and executable once the JIT compiler has produced all instructions. Similarly to the precedent attack, it overwrites the buffer contents once the JIT compilation is triggered lets it compile the payload into the JIT code region.

Attempting to attack the interpreter and the bytecode, Park et al. [9] present an attack on the interpreter through the compilation of malicious bytecode. The attack proceeds in four main phases against SpiderMonkey. The attacker leaks the location of a JavaScript context object and the JavaScript function object of a victim function. The attacker overwrites the function address contained in the function object with the address of a target function (*e.g.* the C library `system` function) that holds the necessary bytecode to hold the payload. The attacker also overwrites the contents of the context object to hold a string representation of the path to the desired program to be executed (*e.g.* `/bin/sh`). The interpreter loads the objects onto the stack (2a) and launches the payload.

Summary: The JIT compiler manipulates and interacts with several security-critical components at run time. JIT IR, JIT data, and JIT code are the input and output of the compiler, and while the output has been the focus of the multiple attacks presented earlier (and therefore the corresponding defenses), the input remains untouched. Moreover, looking at the big picture, the VM also requires bytecode and object tables that are generated at source code compilation and will usually not be modified at run time. It also needs objects that are modified at run time and contain sensitive information such as function pointers or shape and scope metadata. All of these elements are critical to the security of the VM and require particular attention.

2.3 VMs Countermeasures

Defenses applied to VMs started from setting up the standard protections around the JIT code region as it is the main target for attackers and a critical element of the VM run time. The addition of solutions expands more and more other JIT compiler internal structures to make them more resilient. The solutions refine the isolation between the components, shrinking the impact area of attacks targeting one of them. Lian et al. [8] categorize them into three major axes **diversification**, **memory protection**, **capability containment**, to which we add another one using dedicated hardware-supported features, **hardware-enforced isolation**. These categories are mostly presented in chronological order and highlight the interest in finer isolation at a lesser cost.

2.3.1 Diversification

Breaking determinism: Answers to JIT spraying have the main objective of fuzzing the output of the JIT compiler. The deterministic generation of JIT code is what attackers exploit to hide the payload in constants. This determinism is present in the JIT code region through the contents of the JIT code, its layout, and its location in memory. As stated by Forrest et al. [100], *"the role of diversity is to extract robustness against malicious events similarly to biological systems"*. Diversification solutions for the JIT code region can be split into two main categories, as presented by Lian et al. [8]: **intra-instruction** and **code layout randomization**. Additional **location (re)-randomization** with ASLR is discussed in Appendix A and is presented in summary Table 2.1.

Code Contents and Layout Randomization

Motivation: Adding entropy to the JIT compiler output should invalidate attackers' assumptions. Reordering and randomization of instructions, injected at compilation time or even in a self-induced manner at load time, help defend against JIT code injection and reuse. Although the content of the JIT code matters, the layout of its components is also a crucial attack vector following JIT Feng-Shui principles [86], [89].

Register randomization [11], [12], [15] and *Callee-saved register save slot reordering* [101], [102] shuffle register operands in the produced output. This solution comes as an additional compiler pass and adds fixed overhead.

Constant Folding [12], [86], and its counterpart *constant blinding* [6], [12], [13], [15], [86] handle large constants in generated machine code. It runs as a traditional compiler

optimization pass that pre-computes known operations on constants at compile time. *Folding* transforms the remaining of the large user-supplied constants by splitting them into smaller 2-byte values. *Blinding* transforms untrusted user-supplied constants by XORing them with a random value at compile time. At run time, the result is then re-XORed against the random value to reveal the obfuscated constant. These solutions add overhead in code size and execution time.

NOP insertion [12], [13], [86], [102], [103] is the process of adding a random amount of semantic NOPs between legitimate instructions. The accumulation of NOPs makes the location of each instruction harder to predict. Jangda et al. [103] use a list of NOPs with different sizes and semantic NOPs using other instructions that preserve the processor state and minimize the risk of creating new gadgets. The authors use those NOPs and insert them into the generated code to add noise to the result.

Function Permutation [104] randomizes the order of functions and procedure bodies in the code segment and adds high entropy at a low-performance cost to the generated code and data. The Readactor tool [102] uses an indirection layer to hide the code pointers of the randomized functions to prevent any access to the hidden functions, as they are stored in unreadable memory (see Section 2.3.2).

Discussion: Constant folding and blinding (for 3-byte constants and larger) have been implemented in some major web browsers such as Google Chrome or Microsoft Edge as detailed by Gawlik et al. [87]. However, Athanasakis et al. [6] show that attacks are still possible using 2-byte long or shorter constants. They also show that applying the solution to all immediate operands regardless of their size adds an overhead of up to 80% in terms of the number of instructions. Instruction diversification and obfuscation come as a low-cost setup solution to counter basic code-reuse attacks. These solutions add entropy to the generated code and make it harder to use directly by an attacker. The simplicity of the solution still falls short when the attacker knows the obfuscation techniques.

2.3.2 Memory Protection

Enforcing Data Execution Prevention: A common security principle to defend against code injection attacks is to prevent data execution in the stack and on the heap. This defense became famous with the PaX team [105] as Writable XOR eXecutable (W^X) or Data Execution Prevention (DEP) [106], [107]. Enforcing the idea of mutually exclusive permissions is essential to protect applications and systems from malicious injection and execution through the stack or heap. The concept of W^X clashes with the main idea of a

JIT compiler, since it needs to permanently rewrite machine code that has to be executed later on. To still provide guarantees on the generated JIT code, the following techniques are presented: **transient protection** grants temporary rights at run time, and **specific read primitives** use specific hardware-defined memory for the data. The use of several mappings on the same region is presented through process isolation in the next section.

Transient Protection

Motivation: Restricting the permissions of the JIT code following the principle of least privilege handles one part of the `W^X` policy, granting new temporary permissions at run time lets the compiler write or execute a given stub of machine code in JIT code memory.

A way to enforce `W^X` is to grant restricted privileges to the JIT code region and grant the additional privilege at run time. The authors of JITDefender [14] and JITSafe [15] propose an adaptation of the concept on the Tamarin Flash VM. They define the JIT code memory region as `RW` by default. The accesses for execution are instrumented to make the pages executable but not writable (`RX`) then removing the execution permission right after. The resulting overhead is less than 1% but these solutions do not protect against JIT spraying attacks where the control flow vulnerability is triggered from within the JIT code.

Taking a different approach, the authors of JITScope [19] and Readactor [102] authors make the JIT code `RX` by default. JITScope implements three delegates (code routines) to instrument JIT code access. These three delegates serve different purposes, namely `write` gives access with write permission (`RW`) to the JIT code region, `fwd-exec` and `bwd-exec` instrument the call and return from the JIT code and make sure that it can only be accessed through them. As it reprotects memory more frequently, the incurred performance overhead is higher (from 1.6% to 6.0%). Readactor switches between `X`-only pages and `RW` pages at rewrite and installation time and adds an overhead of 4%.

Discussion: While these solutions protect against a single-threaded environment, race condition attacks have been proposed by Song et al. [17] against the V8 VM. They use Web Workers, a mechanism that allows JavaScript programs to spawn multiple threads with communication and syncing capabilities. As an attacker uses one thread to trigger JIT compilation and has access to writable memory, another exploits a vulnerability to corrupt the just-accessed writable memory. While this requires the attacker to predict the address of the memory that will be made writable by the first thread, it defeats the

assumption of `W^X`. A solution through process isolation and double-mapping of memory is presented in the next section.

Specific Read Primitives

Motivation: Some other solutions rely on specific memory permissions to alter the `read` composite of the memory access and thwart the memory disclosure vulnerabilities at the heart of ROP attacks, making it impossible to disassemble the generated JIT code.

Backes et al. [108] use e**X**ecute **n**o **R**ead memory to prevent the JIT code from being read while leaving the memory `WX`. The authors ensure that the code can be executed by the processor but cannot be read as data. The primitive is implemented as a kernel-level modification for both Linux and Windows and is emulated in software at the cost of a run-time overhead of 2.2% and 3.4%, respectively. As specified above, while Readactor [102] uses a transient protection principle, it switches between `X`-only and `RW` permissions over the memory region.

As reading the memory is the first step for a usual ROP exploit, making read executable memory unusable is another way to fix the issue. *Heisenbyte* [109] use destructive code reads to restrict the ability of adversaries to use executable memory that has been exposed through a vulnerability to memory disclosure. They use hardware virtualization support to identify read operations on executable memory and can instrument the JIT code accordingly, forcing the program to crash in case of an attempt to execute a read memory zone. In the same vein, *No Execute After Read (NEAR)* [110] prevents the execution of pages that have been disclosed but have not been applied to JIT compilation as it does not support writable pages.

Discussion: Tweaking the read primitives gives strong guarantees to the underlying memory. However, since they rely on specific hardware, these solutions have not been ported to production VMs. They can guarantee the integrity and privacy of the generated JIT code. Then, the question of the integration of this type of solution in modern VMs and the issues it brings up for the VM itself has to be addressed. In addition, *constructive reloads* [111] have been presented where multiple copies of the native code are added, one of which will be destroyed and another as a mirror to construct the ROP payload.

2.3.3 Capability Containment

Isolation of user-supplied input: Containing and isolating untrusted elements from performing collateral damage (such as pivoting the stack [82], [83]) lowers the impact of attacks. Avoiding side effects from a critical VM component through isolation can occur in two main ways: *process isolation* and *control-flow integrity*. The first method isolates the critical component(s) from the rest of the system, while the second ensures only a correct set of paths can be taken by the application even when dealing with untrusted input. An extension of process isolation comes in the form of *sandboxing* and is presented in Appendix B and Table 2.1.

Process Isolation

Motivation: Isolating the processes of the VM separates the components of the VM and their accesses to sensitive memory regions. Virtual memory makes it possible to get two different virtual mappings with different permissions linked to the same physical memory, effectively monitoring memory accesses.

Without using a sandbox, Jauernig et al. present Lobotomy [16], an architecture that separates the compiler and executor of a JIT engine in as many different processes that share the memory region of the JIT code. The JIT engine is split into two processes, the compiler and executor, that interact through separate execution contexts and synchronize over shared memory. Concretely, the compiler process clones itself into the executor that holds on, waiting for its semaphore. The compiler is triggered when interpretation detects a trace that is ready for compilation. At the end of the compilation, the compiler unblocks the executor to execute the JITted code. For an existing trace, control is given to the executor directly. When blocks and unblocks occur, a *context switch* is performed through two system calls `getcontext` (which saves data structure) and `setcontext` (which resumes from given data). Shared memory allows the different execution contexts to share data and serves two main purposes: (1) double-mapping the JIT code pages (`RW` for the compiler, `RX` for the executor) (2) easy communication between the compiler and executor through mutexes and data. In general, this technique adds a run-time overhead of about 30% (but up to more than 450%).

An industrial application of process isolation without sandboxing has been initiated with *Arbitrary Code Guard (ACG)* in Microsoft Edge [112]. When ACG is applied to a Microsoft Edge Process, it prohibits the allocation of new executable memory or mod-

ification of existing executable memory. As this idea clashes with the concept of JIT compilation, the JIT compilation process is separated from the content process. The isolated process receives JavaScript bytecode and sends back machine code in return. Microsoft now explores the application of this principle in tandem with hardware support for acceleration and integration in the Windows OS.

Discussion: Both solutions isolate the process of JIT compilation from the rest of the VM. The double-mapping of memory guarantees a thread-safe isolation of the JIT code in its writable and executable state. While it enforces strong guarantees on the generated code and its accesses, it comes with a deterring cost in terms of overhead

Instruction Filtering

Prohibition through filtering: Before resulting in precise Control-Flow Integrity (CFI) [52] to add guarantees to the control flow of the program, simpler *instruction filtering* mechanisms prevent critical instruction from executing. At the process level, JITSec [113] proposes a lightweight solution to filter critical operations from the JIT code. The authors decouple *sensitive* from *non-sensitive* code (system calls from normal function calls). They add a kernel module to block the execution of sensitive code from writable memory pages. It is placed between user processes and the system call handler. It launches a small trampoline of assembly code that (1) stores the content of the registers on the stack; (2) calls a policy-enforcing monitor function; and (3) restores the content of registers. Step (2) locates the return address where the control flow is supposed to return after the system call, checks it against the stack and heap memory region and, if it is located in one of them, replaces the system call with a call to `exit`. The solution prevents the execution of system calls at a small performance cost and lowers the impact of code injection. It performs an equivalent of what the Linux `seccomp` [114] solution accomplishes, namely system call filtering, at the cost of 2% performance overhead. It guarantees that no `syscall` is used by an attacker-supplied piece of code, but it does not prevent code reuse attacks in this context.

Control-Flow Integrity

A primer on control-flow integrity: Regarding control-flow integrity (CFI) as defined by Abadi et al. [52], it is a technique that defines security policies that software execution must follow and comply with a predefined Control-Flow Graph (CFG). This

CFG is defined ahead of time by analysis (source code analysis, binary analysis, or execution profiling). CFI requires that, at run time, whenever a machine code instruction transfers control, it jumps to a valid destination determined by the CFG. If most of the instructions target a constant destination, the verification can be performed statically. However, for computed transfers, a dynamic check needs to be performed to ensure the sanity of the destination. Checks should be performed on both *forward-edge* (*i.e.* calls, jumps) and *backward-edge* (*i.e.* returns) transfers. Typical solutions enforce these checks at run time using **tags** and **run-time checks** for *forward-edges* or **shadow stacks** [115] for *backward-edges*. The tag/check goes along with the constructed CFG to verify that a control-flow branching is correct. A shadow stack copies the return addresses on calls and checks the run-time one against the saved one on returns, ensuring that they have not been tampered with by an attacker. The granularity of the solution depends on the precision of the branch identification, and therefore on the precision of the CFG. Intel presents an industrial application of this type of protection as they extended x86 processors with the Intel Control-flow Enforcement Technology (CET) [21, Vol.1 Ch.17] that adds a coarse-grained tagging mechanism to control-flow landing points (`endbranch`) as well as a shadow stack mechanism to check return addresses on return instructions.

Application to VMs: Porting this type of solution to dynamically generated machine code such as JIT-compiled code is a challenge, especially due to the impact on performance the CFI enforcement techniques usually incur. Niu et al. present RockJIT [18], a solution that enforces modular control-flow integrity (MCFI) [116] for JIT compilers. MCFI is a CFI technique that allows modules to be independently instrumented and linked statically or dynamically. They address the security aspect of both the JIT compiler (static code) and the JIT code region (dynamic code) with different levels of CFI enforcement. In the first case, the JIT compiler is compiled and instrumented to generate an MCFI module. RockJIT then generates a CFG based on the information in the module, and constructs MCFI tables for later use, that define this graph before launching the execution. For VM, the CFG generation strategy is relatively *fine-grained* and performed offline. In comparison, the CFG for the JIT code is *coarse-grained*, as it compares the information emitted by the compiler at the time of indirect branch generation with a set of sane targets. To determine if the destination of a control-flow transfer is sane, RockJIT uses a verifier that keeps three address sets populated at machine code generation: start addresses, indirect branch targets, and direct branch targets. They are updated after code installation, deletion, or modification. This way, the JIT compiler itself runs under

a fine-grained CFI policy and JIT code a coarse-grained policy (to reduce the impact on a performance-critical component). The authors argue that this level of protection is enough to protect the JIT code since it cannot contain dangerous instructions such as system calls. In general, the solution incurs around 15% of performance overhead.

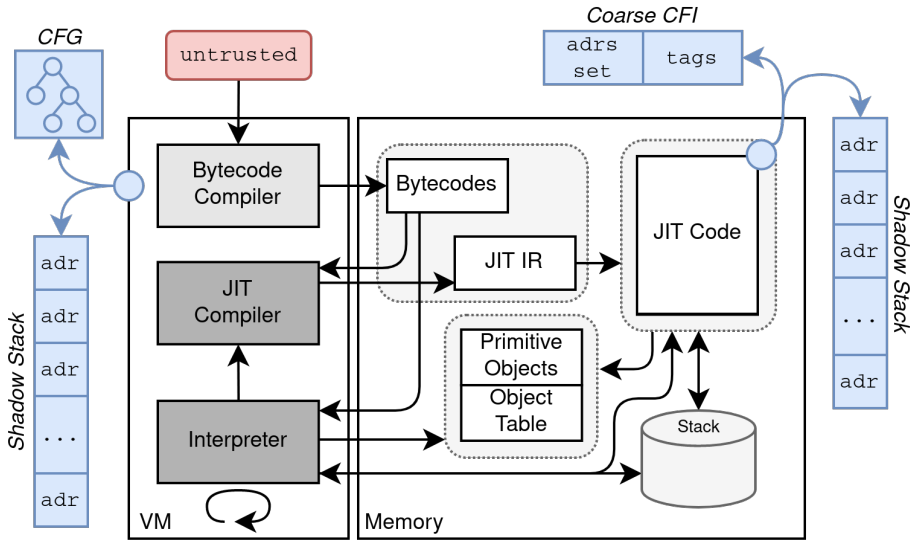


Figure 2.8 – CFI solutions.

JITScope [19] authors present a similar idea of mixed-level CFI enforcement. The solution uses the type information from the source code to deploy fine-grained CFI on the statically compiled code. For each function, it computes an ID derived from the function type information and instruments a CFI security check to match the transfer target ID against the expected ID. For the JIT code, a JavaScript delegate called `CodeGen` is added to instrument a random ID at compile time, before the generation of the machine code of an input function. When the JIT compiler is triggered to generate an indirect call or jump, the delegate instruments CFI security checks in the generated code before the call or jump. JITScope validates the existence of the ID before the target function and ensures that the transfer target is a valid JIT function entry. In addition to *forward-edge* transfers protection, the *backward-edge* transfers are protected with a shadow stack. At each entry of the function, it stores the return address of the current function frame. At function exit, it checks the return address of the main stack against the one stored in the shadow stack. If they do not match the stack has been tampered with, and the program should raise an exception. The stack is set in a separate memory region, indexed by a dedicated segment register (available in the x86 architecture and not accessible through

normal instructions). A new stack is also built for each thread, using thread-local storage to guarantee a thread-safe shadow stack. In general, the solution adds around 10% of performance overhead.

Discussion: Both solutions are summarized in Figure 2.8, the static code is protected using a CFG (for forward-edge checks) and a shadow stack (for backward-edge checks). Even though the dynamic code is protected using a shadow stack, it is not possible (and too performance-impactful) to extract a precise CFG of JIT code at run time. A degraded CFG version is used, either through function tagging or using a restricted set of possible addresses. Overall, CFI enforcement solutions protect against complex attacks, but add an important performance overhead.

2.3.4 Hardware-Enforced Isolation

Extracting hardware performance: Modern computing platforms are now diverse in terms of architectures, software provisioning models, and the type of applications they support. Data confidentiality and integrity are now concerns not only from attacks over the network but also from attacks originating from software or hardware components on the same platform. This requirement comes from systems that hold multiple mutually untrusted software components, *e.g.* that share a cloud platform, or handle code/data belonging to different users. It extends to any computation that involves security-critical components or sensitive data. The answers to these memory isolation requirements can come in the form of hardware-based solutions such as **Memory Protection Keys (MPK)** or **Trusted Execution Environments (TEE)**. We present here the application of these techniques to a JIT compiler as found in the literature.

Trusted Execution Environments

A primer on trusted execution environments: Trusted Execution Environments (TEEs) provide hardware-level guarantees. Schneider et al. [117] categorize them in four levels: (1) verifiable launch of the execution environment for sensitive code/data in a way that can be verified by a remote entity; (2) run-time isolation to enforce confidentiality and integrity of sensitive code/data; (3) trusted IO to enable secure access to peripherals/accelerators and (4) secure storage for TEE data that must be stored persistently and made available later on only to authorized entities. Intel presented an industrial TEE named *Software Guard eXtension (SGX)* [118] added to Intel CPUs, that was first intro-

duced in 2015 with the sixth generation of microprocessors. They allow isolated execution environments called *enclaves*, encrypted regions, created within a user-mode process, that cannot be accessed by any (higher privileged) system entity. The enclave is decrypted on the fly by the CPU using an enclave-specific key. Before the enclave memory is loaded onto the CPU, SGX verifies its integrity to ensure that an attacker does not tamper with the contents.

Application to VMs: JITGuard [10] authors based their defense solution around an SGX enclave, which overview of the design is presented in Figure 2.9. Since interacting with the contents of an SGX enclave adds considerable overhead, especially through context switches, isolating the whole JIT engine along with the JIT code region in an enclave results in a huge amount of performance overhead. In particular, since the JIT code frequently interacts with static code and is called transparently, it cannot reside in an enclave. Therefore, JITGuard adds three important security principles to its design: (1) isolate the JIT compiler and its data in an enclave, (2) randomize addresses of the JIT code and JIT stack memory, and (3) build an indirection layer for trampolines (transitions between JIT and static code).

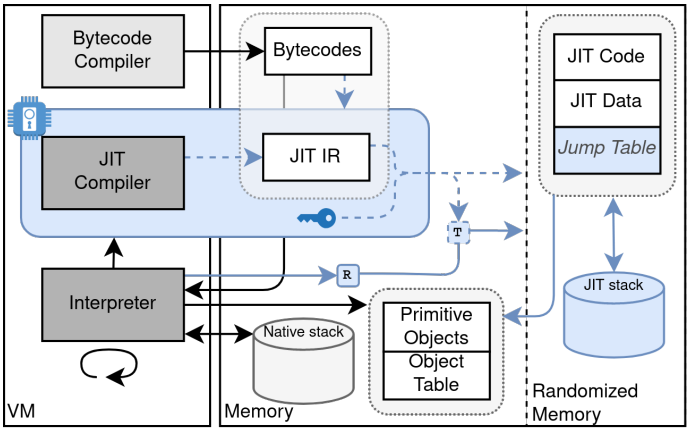


Figure 2.9 – JITGuard solution.

The JITGuard initialization step allocates two memory regions, the trampolines and JITGuard-region where the JIT code, the JIT stack and the writable mapping of the trampolines reside. The enclave (1) is then started containing both the JIT compiler and a secret to lead to the location of the JITGuard-region. The randomization (2) of the JITGuard-region is maintained by mediating all memory accesses to the JITGuard-region through the enclave (for JIT code generation, modification, or deletion). The indirection (3) needed by trampolines (`T` on the figure) is achieved with a double-mapping of them,

one executable, the other writable and used only through the JIT compiler, as it is hidden in the JITGuard-region. To enable efficient interactions between static and JIT code without involving the enclave for every interaction, trampolines need fast access to JITted functions. JITGuard establishes an indirection mechanism using an offset over a segmentation register (`R` on the figure), the contents of this register are available only through a system call. To ensure a clean separation between JIT code and static code, JITGuard employs its own stack, which is stored in the JITGuard region. In summary, the complete JITGuard solution introduces a performance overhead of 9.8%.

Discussion: TEEs provide hardware-enforced guarantees on the execution of software. However, Intel SGX was discontinued in 2021 due to the list of vulnerabilities that have been found against the solution (mostly side-channel attacks). *SGAxe* [119] authors present a transient execution attack that can recover SGX attestation keys from a fully updated Intel machine trusted by their attestation server. *SmashEx* [120] authors show that the asynchronous exception handling mechanism is prone to reentry vulnerabilities.

Memory Protection Keys

A primer on memory protection keys: Another hardware-supported memory isolation solution is the use of *Memory Protection Keys (MPKs)*. This mechanism provides efficient and secure in-process isolation. MPKs come as an extension to page-based memory permissions, allowing changes in the permissions of memory ranges without going through the kernel-level modification of the page, which is usually slow. They instead tag page-table entries with a protection key and store the permissions for these keys separately. Each memory region can have one associated key, while processes can have one or more via special registers. Careful access to these keys allows for in-process isolation. Implementations differ in the number of keys, the types of permissions and the granularity of the memory region they propose [31], [121]–[123]. Intel's *Memory Protection Keys (MPK)* [21, Vol.3 Sec.4.6] use 4-bit keys stored in the page-table entry. This allows for 16 different domains per process. To change permissions, the program uses an unprivileged instruction to write to the local thread key register *User Page Key Register (PKRU)*. As the PKRU is non-privileged, it allows for fast domain-switching in the userspace. The Linux kernel provides support for Intel MPK (since v4.6) through three system calls: `pkey_alloc`, `pkey_free` and `pkey_mprotect`. The kernel maintains a bitmap to keep track of the allocated keys.

Application to VMs: Libmpk [124] authors have extended the Intel solution with

a software abstraction to virtualize the protection keys. They applied their solution to three JavaScript VMs (SpiderMonkey, ChakraCore, and V8) and show that they can enforce `W^X` on the generated JIT code. Their objective was to protect the VM against race conditions [17]. The authors presented two different techniques to implement their solution: "*one key per page*" and "*one key per process*". In the first method, they replaced permission switches with MPK equivalents, resulting in a faster, context-free solution. The second method utilized a single protection key to safeguard all JIT code regions. It ensured that only one thread had access to this protection key at any given time.

NoJITsu [9] authors extend the usage of protection keys and design their solution around Intel's MPK, an overview is provided in Figure 2.10. They compartmentalize the JavaScript engine following three axes: (1) separate JIT data from JIT code and give them different permissions, (2) add a routine to allow temporary accesses using conjoined calls to `set_pkru`/`write_pkru`/`recover_pkru` to change the value of the `PKRU`, and (3) isolate sensitive objects such as sensitive types (script, shape, function, etc.) and primitive types (scalar and array data types) from other objects. Overall, their strategy lies in a fine isolation technique of critical elements between each other. All memory write instructions are instrumented to use the appropriate run-time memory permissions based on which data types the instruction may access. Instrumentation targets are determined using dynamic object-flow analysis to detect accessor functions (functions that directly write to JavaScript objects). Accessor functions are grouped into four different types, either *member accessors*, member functions of a JavaScript object class that write to private variables; *payload accessors*, member functions that update the actual payload of a JavaScript object; *initialization accessors* that initialize JavaScript objects; and *garbage collection accessors* that update information in objects metadata and objects metadata to ease later garbage collection. Overall, the solution adds a 5% performance overhead.

Discussion: MPKs provide fine-grained intra-process isolation. However, several issues prevent the widespread usage of Intel's implementation. First, the built-in number of keys is not sufficient for fully fledged applications (such as OpenSSL [124] or persistent memory objects [123]). Also, use-after-free vulnerabilities have been discovered on the mechanism [124]. Other implementations compensate for these problems and provide comparable capabilities [31], [121]–[123] to enforce strong properties on the JIT code region.

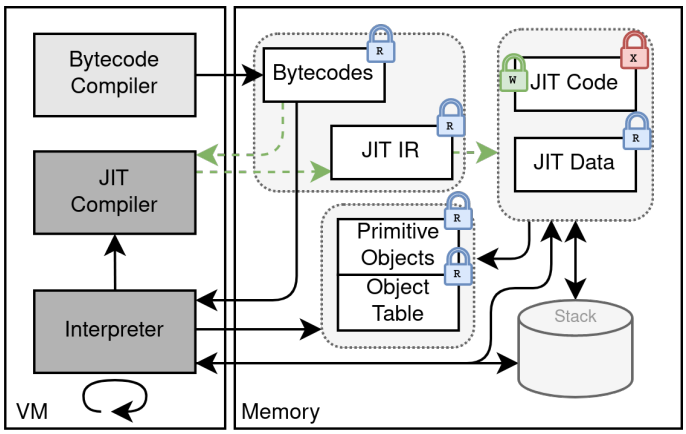


Figure 2.10 – NOJITsu design.

2.4 Summary

Attack Methods: Overall, attacks against VMs have started by focusing on the JIT compiler and the tightly linked JIT code region. Its native access to memory, with all permissions and processing of untrusted and unverified inputs, makes it a critical component. Injection of either a version of the full exploit or gadgets through the input source code lets attackers weaponize their exploit using the built-in compiler. Finally, securing the compilation process and code execution does not suffice if the inputs can be tampered with by an attacker. The data itself has been an attack vector either through bytecode or JIT intermediate representation, allowing the spraying of shellcode in memory even with active protections on the VM, particularly on the JIT compilers and JIT code region.

Expected Capabilities: For a successful attack, an attacker needs access to certain memory vulnerabilities. Typically, a method to corrupt a code pointer leading to an attacker-controlled memory region or a stack pivot is necessary to execute code injection or transfer control to a Return-Oriented Programming (ROP) chain. In the context of code-reuse and data-only attacks, a memory disclosure vulnerability is necessary to determine either the location of the binary in memory or the data memory for extracting gadgets. Assuming the existence of such vulnerabilities is common in browser exploits and should, therefore, be the norm when designing defenses around VMs. We define these vulnerabilities and capabilities as:

C1: An attacker can supply arbitrary input source code to the VM, that can later be interpreted and recompiled at run time.

C2: An attacker has access to a memory corruption vulnerability to redirect the control-flow of the program either from the statically-compiled VM or the JIT-compiled code.

C3: An attacker has access to a memory disclosure vulnerability to determine the location of the JIT code or sensitive components.

Table Summary: The Table 2.1 presents the main defense solutions that have been deployed around VMs, classified around their main defense mechanism. The VM the solution is deployed on is also presented, usually a JavaScript browser VM such as V8 or SpiderMonkey. The effect on attacks is tagged as countered, partially countered, or inefficient. It is also important to note that all defenses are based on x86 architectures. $\sim$ denotes that the chosen solution does not fully protect against the corresponding attack. Diversification makes it more difficult to carry out a code reuse attack, but it does not completely prevent such attacks. Sandboxing solutions [17], [125] decouple the access to writable and executable memory and provide a certain level of code verification. However, they do not fully prevent code reuse attacks. Instruction filtering [113] prevents explicit `syscalls` but does not provide complete protection against attacks themselves. Lastly, JITGuard [10] is effective against JIT IR data-only attacks but may not fully protect against bytecode data-only attacks.

Defense Methods: As software security evolves, defenses around VMs have initially focused on disrupting the determinism introduced by the JIT compiler. Properties like diversity and randomization play pivotal roles in fortifying the security of the JIT compilation process, particularly against simple injection attacks. However, these measures may prove inadequate against more sophisticated threats like code-reuse or data-only attacks. Tightening memory permissions stands out as a straightforward method to guard against code injection, preventing attackers from writing to executable memory. Given the explicit need for executable memory by the JIT compiler, double mapping of the region emerges as the optimal strategy, thwarting race-condition attacks. While code reuse remains a concern, disassembling the JIT code exposes available gadgets to attackers. Two protective measures include removing the readability of the executable JIT code and

Name	Defense mechanism	VM	Code Injection [4], [8], [85]	Code Reuse [5]–[7]	Data Injection [9], [10]
INSeRT [11]	Diversification	V8	✓	✗	✗
RIM [12]	Diversification	Tamarin	✓	✗	✗
librando [13]	Diversification	Hotspot, V8	✓	✗	✗
- [103]	(Re)-Diversification	JikesRVM	✓	∼	✗
JITDefender [14]	Transient Protection	Tamarin, V8, JSCore	✓	✗	✗
JITSafe [15]	Transient Protection, Diversification	Tamarin, V8, JSCore	✓	✗	✗
XnR [108]	XnR	Application-Agnostic	✗	✓	✗
Readactor [102]	XnR, Diversification	V8	✓	✓	✗
Lobotomy [16]	Process Isolation	SpiderMonkey	✓	✗	✗
ACG [112]	Process Isolation	Microsoft Edge	✓	✗	✗
NaCl [125]	Sandboxing	V8, Mono Runtime Engine	✓	∼	✗
SDCG [17]	Sandboxing	V8	✓	∼	✗
JITSec [113]	`syscall` Filtering	Application-Agnostic	∼	∼	✗
JITScope [19]	CFI, Transient Protection	SpiderMonkey	✓	✓	✗
RockJIT [18]	CFI	V8	✓	✓	✗
JITGuard [10]	Intel SGX Enclave	SpiderMonkey	✓	✓	∼
Libmpk [124]	Intel MPK for Transient Protection	SpiderMonkey, ChakraCore, V8	✓	✗	✗
NoJITsu [9]	Intel MPK for Fine-Grain Isolation	SpiderMonkey	✓	✓	✓

Table 2.1 – VM defense solutions.

implementing a robust control-flow integrity mechanism. Fine-grain, restrictive isolation between the components gives the most reliability to the VM but comes with a net impact on performance.

Defense Requirements: The coarse-grained features of the software and hardware (*i.e.* virtual memory, privilege modes, *etc.*) are not sufficient to protect against run-time attacks. On the other hand, the performance of fine-grained software solutions is deterring implementation in production environments. A clear need for hardware-accelerated fine-grained isolation is highlighted by convoluted run-time attacks and the complexity of VMs and their component. We define expected *"software requirements (SR)"* as:

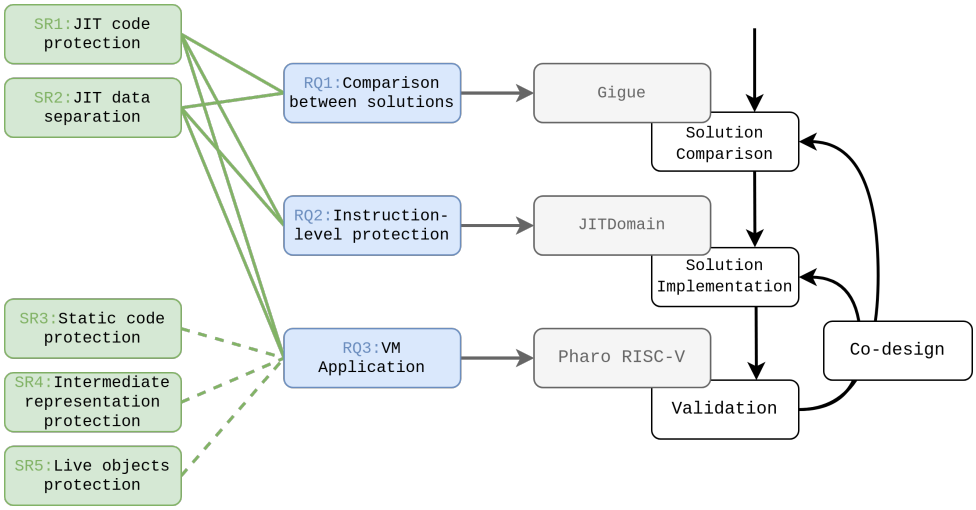
SR1: The JIT code should be writable only at generation and installation time, then executable only. It should implement a (most precise possible) CFI solution along with a `syscall` filter.

SR2: The JIT data should be separated from the JIT code, should be accessible only through the associated JIT code and should not be executable.

SR3: The static code of the VM is a source of gadgets that should be compiled offline with controls on precise CFI and/or gadget remover tools.

SR4: The intermediate representation (bytecode or JIT IR) should only be writable at installation, from a single entry point in the VM.

SR5: Sensitive live objects should be given restricted write access to their attributes to avoid type confusion and arbitrary memory writes.



These requirements are composable and cover the known sensible points in a VM runtime. In the context of this thesis, we will focus on the JIT code region in particular (SR1/SR2) but notice the importance of other requirements, leaving them for future works. We extend the previously introduced research questions with links to their corresponding software requirements in the figure above. Moreover, in addition to these requirements themselves, the means to enforce them are equally important. As the latest defenses are built on top of solutions that present known vulnerabilities, the transparency and flexibility of the implementation alter their adoption.

Chapter 3

Open Hardware-Accelerated Security Features

As discussed in the previous section, contemporary approaches to safeguarding in defending performance-critical components involve leveraging hardware-supported features to seamlessly incorporate robust security measures during execution, all without introducing significant time overhead. Despite these efforts, security solutions embedded in vendor hardware often derive from alternative mechanisms (such as tracing or debugging functionalities) or are directly provided by the vendor, offering minimal flexibility for post-deployment modifications. Given these constraints, we advocate the RISC-V Instruction Set Architecture (ISA) as an optimal choice for open prototyping and the development of adaptable security solutions. The ongoing evolution of hardware and software within the RISC-V ecosystem is noteworthy. Recent attacks, reminiscent of those outlined in the preceding chapter, have been successfully executed. These incidents emphasize the critical need for open and adaptable security solutions.

3.1 Motivation

The motivation for dedicated open hardware security features is twofold. First, guarantee the efficiency of the solution through a dedicated set of features, either through an extension of the processor itself or the presence of a dedicated coprocessor; second, ensure the soundness and maintainability of the solution, either through its open-source nature or flexibility in the design [126].

Need for dedicated solutions: Defining hardware-supported security solutions over vendor architectures suffers from the lack of dedicated information extraction. Several solutions have extended the debugging modules of the cores to define a security policy. Intel Last Branch Record (LBR) [21, Vol.3 Ch.19] records the history of the most recent N branches, with N equal to 4, 8, 16, or 32 depending on the processor. It was the basis for

a corpus of work [127]–[129] using the LBR as a shadow stack, but the limitation of the number of stored addresses made these solutions vulnerable to history-flushing attacks [130], [131] hiding the effects of a ROP attack behind N genuine branches. Modern processors also provide extensions to capture debugging information. Intel Processor Trace (PT) [21, Chapter 33] and ARM CoreSight [132] provide traces for offline post-processing debugging. Recent work uses these hardware extensions to enforce Control-Flow Integrity (CFI) [133]–[137] or Dynamic Information Flow Tracking (DIFT) [138]. Extending these mechanisms to enforce security policies at runtime requires additional decoding features [139] for better performance.

Need for flexible solutions: Industry has added efficient dedicated hardware-assisted security extensions. Among them, when considering Intel, were added: Intel Trusted Execution Technology (TXT) [140], Intel Memory Protection Extensions (MPX) [21, Appendix E], Intel Software Guard Extensions (SGX) [21, Vol.3 Ch.34-39], Intel Control-Flow Enforcement Technology (CET) [21, Vol.1 Ch.17], Intel Memory Protection Keys (MPK) [21, Vol.3 Sec.4.6], and many more. As they are deployed by the vendor directly in silicon, any problem in the design or implementation of these solutions may require in-depth modifications in the next generation of the processors, hindering their usage in the current one. For example, Intel introduced MPX in 2013, a hardware-assisted extension for spatial memory safety that performs bound checking, which was found to be considerably slower (up to 4x in the worst case) than its software counterparts. This led to its removal from major compiler toolchains, before being discontinued by Intel in 2019. As another example, Intel SGX was introduced in 2015 to define a trusted execution environment that encrypts its contents and performs on-the-fly decryption within the CPU. It was found to be vulnerable to multiple side-channel and information leakage attacks [119], [120] that resulted in the deprecation of SGX in 2021, from its newer generation of processors. ARM TrustZone [55] was also found to be vulnerable to similar classes of attacks [141], [142]. As a final example, Intel MPK [21, Vol.3 Sec.4.6] was presented in 2019 as an intra-process isolation technique that would avoid costly kernel-level switches from trusted to untrusted domains. It was shown to have security [31], [121], [122] and scalability [123], [124] drawbacks.

A dedicated security platform allows the developer to extract the needed information from the core and to efficiently enforce its security model. The need for prototyping and flexibility before integration into a commercial core brings attention to flexible solutions that are available for multiple purposes and defined on open-source hardware.

3.2 A Primer on the RISC-V ISA

RISC-V [20], [56] is an open, modular, and extensible Instruction Set Architecture (ISA) that gains interest from research and industry as several processors and toolchains become available as an alternative to vendor processors. Available RISC-V tools include a GNU Compiler Collection (GCC) toolchain, an LLVM toolchain, a QEMU simulator, and operating support from Linux and BSD variations. RISC-V architectures are pushed by Chinese industries as an alternative to proprietary ARM processors, such as AliBaba cloud's Wujian 600 development board.

3.2.1 Modularity

The RISC-V ISA provides 32-bit, 64-bit, and 128-bit variants of its instruction set, along with extensions defined in modular groups. The basic set of instructions is defined as the integer extension and available as `RV32I` or `RV64I` (a superset of the 32-bit version). It also defines a set of composable standard extensions that support a fully-featured Operating System (OS). `RV64G`, an abbreviation for `RV64IMAFDC`, contains the integer instructions (`RV64I`), multiplications and divisions (`M`), atomic instructions (`A`), floating-point operations (`F`), double integers (`D`), and compressed instructions (`C`). Processors that implement this set of extensions support the Linux OS. The scope of RISC-V is not limited to these standard extensions; for example, the `E` extension for embedded devices uses a reduced number of registers, and the `V` extension enables vector operations. Other extensions are still being ratified and aim at specific application cases. Among them, the `J` extension for dynamically translated languages and virtual machines is still open to contributions.

3.2.2 Extensibility

The RISC-V standard also defines four custom opcodes, `custom0-3`, which are reserved for custom use and will not be redefined by standard extensions in the future. The primary objective is to ensure that any extension of the ISA restricted to these instructions remains valid in the future, thus providing developers with a stable API for custom functionality prototyping.

In addition to the custom instructions as a whole, *hints* enable the embedding of behavior in existing instructions by passing specific parameters [20, Sec.2.9]. These pa-

rameters do not alter the semantics of the instructions and do not affect a core that does not implement them. However, they can trigger side microarchitectural effects with the available bit space. Although their standard use has not yet been defined for all of them, 7 of them are reserved for custom use in the `RV32I` instruction set. They should eventually include aspects such as spatial and temporal locality of the memory system, branch prediction, security tags, or instrumentation flags.

The Rocket processor [39] integrates these instructions and defines the Rocket Custom Coprocessor (RoCC) interface. This interface acts as an API for custom instructions and works as a programmable `command/response` module that processes custom instructions with direct L1 cache access. It provides a simpler standardized way to extend custom instructions through a coprocessor. The CVA6 processor [3] defines CV-X-IF, an interface to extend the base RISC-V instruction set through a coprocessor.

3.2.3 Memory System

Privilege modes: Three privilege modes are defined in the unprivileged [20] and privileged [56] ISA. The machine (`M`) mode is the highest privilege mode, mandatory for any RISC-V core. Supervisor (`S`) mode is employed by the operating system, and user (`U`) mode is available for the execution of the application. In addition, there is a hypervisor (`H`) mode, the specification of which is still under development. This mode is expected to enhance virtualization and containerization capabilities. These three primary privilege levels segregate the executed code based on the permissions associated with their respective roles. Each hardware thread runs in one of these privilege modes at any given time, with the corresponding rights and control status registers (CSRs).

Paging: In `RV64`, RISC-V introduces two paging mechanisms and virtual memory systems known as Sv39 and Sv48 [56]. Sv39 defines a 39-bit address space, while Sv48 defines a 48-bit address space, both divided into 4KB memory pages. The standard outlines the mapping between virtual memory and the corresponding physical memory using Page Table Entries (PTEs). Each PTE contains the address of the physical frame, along with read (`R`), write (`W`), and execute (`X`) permissions for the current page. In addition to the three permission bits and the address mapping (either 39 or 48 bits), a 10-bit space is reserved in the PTE for future use and research experiments [56].

PMP: The RISC-V standards introduce a hardware Physical Memory Protection (PMP) module [56] designed to enforce permissions on memory access at the hardware level. This module effectively manages various memory regions by specifying address

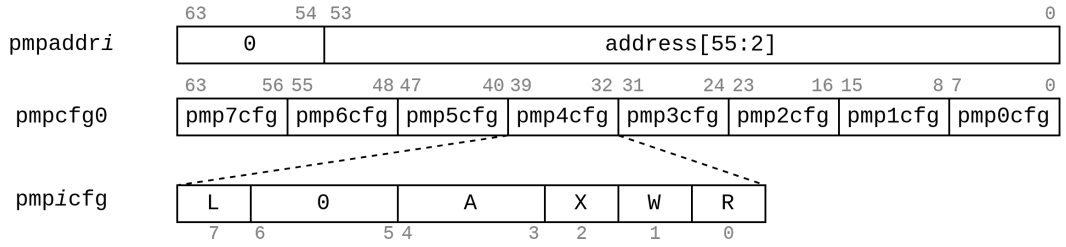


Figure 3.1 – Control Status Registers used by the PMP module.

ranges and their associated permission rights, all configured in machine mode. Providing a per-hardware-thread (hart) perspective, it enables machine mode to control the physical addresses available for supervisor and user mode software. Several CSRs store PMP region information, as illustrated in Figure 3.1. Specifically, the `pmpaddr`i CSRs contain the base address to match the PMP region along with the `A` field of the `pmpcfg`i register.

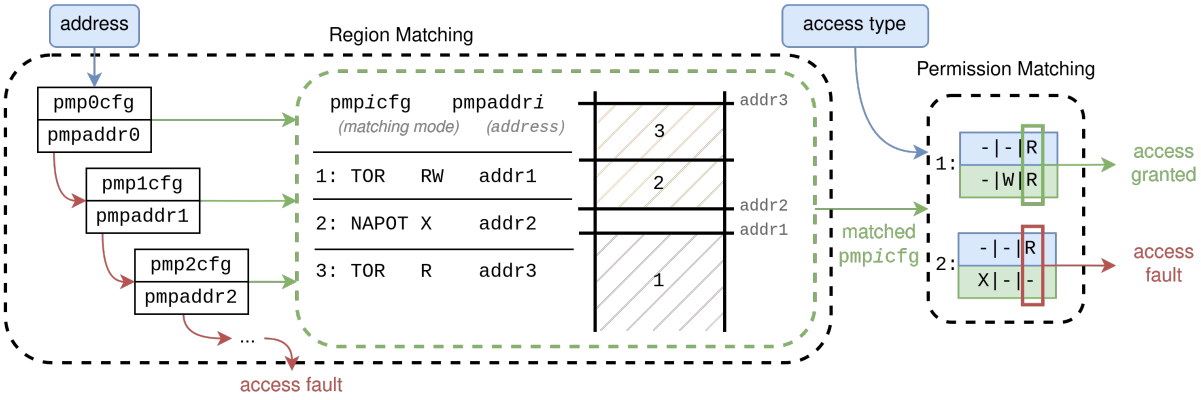


Figure 3.2 – PMP matching mechanism, with region permission matching.

The matching logic to determine whether an input tuple `(address, access type)` is granted access or not is presented in Figure 3.2. The PMP module initially iterates through pairs of addresses `pmpaddr`i and configurations `pmp`i`cfg` to identify the region that matches the incoming address request. PMP regions are matched based on a combination of the base address in `pmpaddr`i and the range type defined in the `pmp`i`cfg.A` field. The `A` field can hold one of four values: `OFF` (indicating that the region is not used), Top Of Range (`TOR`, matching all addresses below it and above the previous region), Naturally Aligned Power Of Two (`NAPOT`, matching a base address and a specified alignment range), and `NA4` (a specific case of `NAPOT` with a length of 4 bytes). In the dotted green region

of the Figure, three regions are defined: region 1 is defined as read/write in `TOR` mode, it matches all addresses beneath `addr1`; region 2 is defined as execute-only in `NAPOT` mode, it matches addresses between `addr2` and the range embedded in the address through its alignment; and region 3 is defined as read-only in `TOR` mode, matching all addresses between the top of the previous region and `addr3`. Once the input address matches a PMP region, the required access type is checked against the permissions granted for that region, which are stored in the last 3 bits of the `pmp`i`cfg` register. Any failure to match a region or access it with incorrect rights results in an access fault.

This module enforces basic access permissions for embedded devices that do not implement virtual memory and is designed to complement paging. It has been used to provide lightweight security solutions [33], [34] and serves as a foundation for a Trusted Computing Base (TCB) [143].

3.2.4 Open-source Processor Implementations

The modularity of RISC-V has driven the creation of a wide variety of core implementations, including compact embedded cores and high-performance Systems-on-Chips (SoCs). A notable example is the PULP initiative, which defines an extensive range of RISC-V cores and core clusters. This spectrum includes the 32-bit 1-stage Snitch [144] core and extends to the Hero [145] multi-cluster heterogeneous accelerators. Application-class processors within this spectrum typically provide support for UNIX-based operating systems and necessitate the implementation of various features such as different privilege modes, atomic memory operations (AMOs), and hardware support for virtual memory translation. Our work focuses on this category of processors since any language VM relies on the underlying operating system for functionalities like memory allocation and interfacing. Prominent implementations in the literature include CVA6 [3], Rocket [39], BOOM [146], and SHAKTI [147]. D"orflinger et al. [148] conduct a comparative survey of these application-class cores. They classify them based on the hardware description language in which they are written, the frameworks supporting their integration, and their performance, area, and power metrics when deployed on FPGAs or ASICs.

3.3 Portability of known attacks to RISC-V

The open-source and modular design of the RISC-V ISA makes it a suitable choice for experimenting with and creating hardware-supported security solutions. Since the ISA includes a limited set of instructions, it raises questions about the potential for attacks presented in the previous chapter to be easily portable.

Cloosters et al. [96] categorize five elements that make it more challenging to determine ROP gadgets on RISC-V and ARM64 (rather than x86 or even ARM32): (1) The program counter register is not a general-purpose register; (2) There is no stack-based return instruction; (3) The arguments are passed to functions via dedicated registers rather than via the stack; (4) Memory alignment prevents execution of hidden unaligned instruction sequences; and (5) Long function prologue/epilogue sequences introduce side effects.

The authors [96] also introduce RiscyROP, a gadget-finding and chaining tool based on symbolic execution, built on top of the `angr` framework [149], which effectively addresses the previously mentioned challenges. RiscyROP scans through a binary to extract interesting gadgets and their information, then chains them into a Return-Oriented Programming (ROP) sequence capable of handling attacker-supplied data. Jaloyan et al. [95] extend the Galileo algorithm [88] to consider, not only the main execution path, but also a *hidden* execution path. They introduce two hidden instruction patterns: overlapping 32-bit instructions composed by using the last 16 bits of one instruction and the first 16 bits of the next one, and two compressed 16-bit instructions forming a genuine 32-bit instruction. This mechanism allows attackers to conceal a backdoor in a RISC-V binary. Gilles et al. [150] present a Jump-Oriented Programming (JOP) attack designed to perform a system call on a system with a limited attack surface, supporting only RV32I architecture. They identify gadgets ending with indirect jumps to create a JOP chain (similar to a ROP chain but ending with `jalr` instead of `ret`) that sets up three argument registers to target a private key, its size, and file descriptor before performing a `write` system call.

Only a handful of JIT compilers have been adapted to RISC-V [151], and this has only been done since 2021. Nevertheless, sophisticated attacks against the ISA have already been developed and are likely to target the corresponding JIT compilers soon. As the RISC-V software environment continues to expand and more VMs are ported to RISC-V [152], it is essential to anticipate these attacks and create safeguards for this architecture.

3.4 Hardware Run-time Protections

Solution Refinement: Jan-Erik Ekberg presents in its lecture [153] the shift to hardware-supported fine-grain run-time protections as a way to gain both preciseness for applications that coarse-grained hardware implementations are not able to provide; and performance from hardware support, that fine-grained software implementations are lacking (grey arrow in Figure 3.3). RISC-V has been an important prototyping ground for upcoming security solutions due to the reserved custom space and availability of open-source components. Tao Lu conducted an extensive survey [154] of the security mechanisms available at the time of writing.

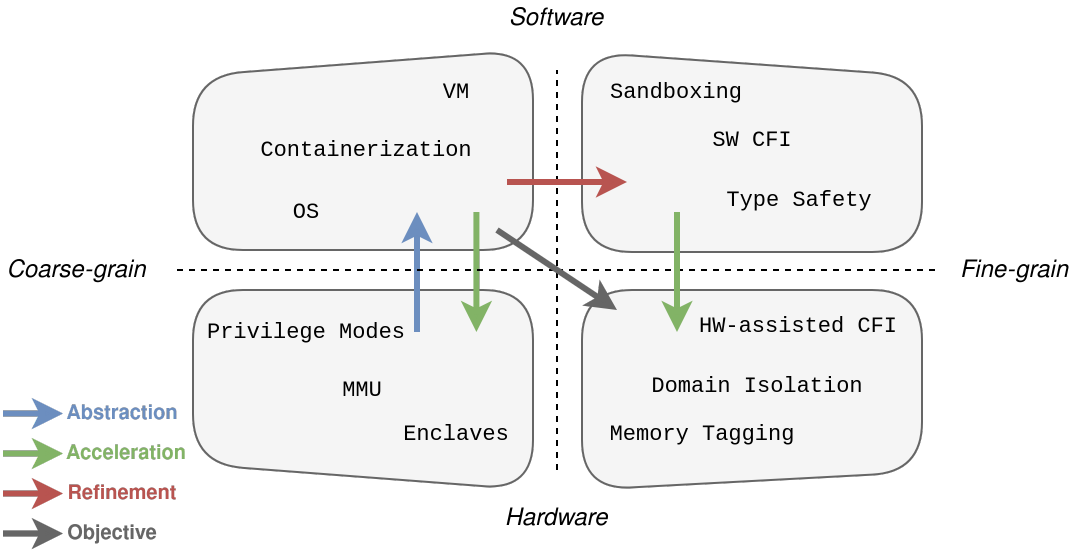


Figure 3.3 – Coarseness of hardware/software defenses.

Coarse-grained Solutions: The global run-time protection and memory isolation mechanisms can be categorized by coarseness, along with their hardware or software implementation. Coarse-grained software solutions, such as virtual memory, the presence of an OS, or containerization (and in a sense VMs) have abstracted underlying hardware features (blue arrow in the figure). In parallel and to support established software utilities, coarse-grained hardware solutions took into account the upper software part. Among them, is the presence of a dedicated Memory Management Unit (MMU) along a Page Table Walker (PTW) to quickly handle virtual-to-physical memory translation (left green arrow in the figure). As presented earlier, RISC-V defines different privilege modes and

exception levels to provide the basis of an OS. As an answer for more isolation, Trusted Execution Environments (TEEs) or enclaves emerged, still keeping the precision coarse, though. A notable example of RISC-V would be Keystone [143] developed open-source along with its Software Development Kit (SDK).

Fine-grained Solutions: The scope of the security primitives now needs to be refined to guarantee the protection of user data and the computing system itself against attackers. This refinement is first presented through software primitives in the paradigms of the code and its associated data with, for example, type-safe languages. But it also comes through the application of guarantees on a binary, such as CFI, which is expensive to implement precisely and verify at run time (red arrow in the figure). The cost in performance of using software-only solutions limits the adoption of strong solutions. Co-designed with hardware support, solutions that accelerate software-defined solutions, or enforce more complex guarantees on the executable code pave the way for future run-time protections (right green arrow in the figure). We present such solutions and RISC-V alternatives in the remainder of this section. We split them into three categories: **hardware control-flow integrity**, **pointer extension**, and **domain isolation**.

3.4.1 Hardware Control-Flow Integrity

Backward-edge protection: Regarding Control-Flow Integrity (CFI) *backward-edge* protections, shadow stacks, despite being a low-cost solution, provide strong run-time guarantees on the code. We present an example in Figure 3.4, where return addresses of the application represented in the center are stored in a separate part of memory. In their study, Burow et al. [115] compare shadow-stack implementations and advocate in the defense of a hardware version that would then become Intel CET [21, Vol.1 Ch.17] or ARMv9 Guarded Control Stack (GCS) [155]. Those implementations dedicate a part of memory to the shadow stack itself, operated in hardware, and not accessible to the user. Several implementations coexist over the RISC-V ISA. FIXER [22] uses a coprocessor and custom instructions to `push`/`pop` return addresses in the deported shadow stack. PHMon [156] is a configurable hardware monitor that uses Match Units (MUs) to react on a succession of instructions and offloads a trigger to a corresponding action through Action Units (AUs). It can be configured to act on calls and returns and offload the return address to its dedicated memory, performing the associated equality check for a shadow stack implementation. Both act as a supplement to the effective call stack and require a check on returns. RIMI [33] defines a hardware domain (memory range) as its

only call stack memory, accessible only through dedicated store instruction and isolates the call stack entirely (presented in more detail in the next subsections). These solutions are a low-cost high-value mechanism, that guarantees the correct control-flow for return addresses. The hardware storage of the shadow stack separates it from the main data memory and sanitizes its access points through custom instructions.

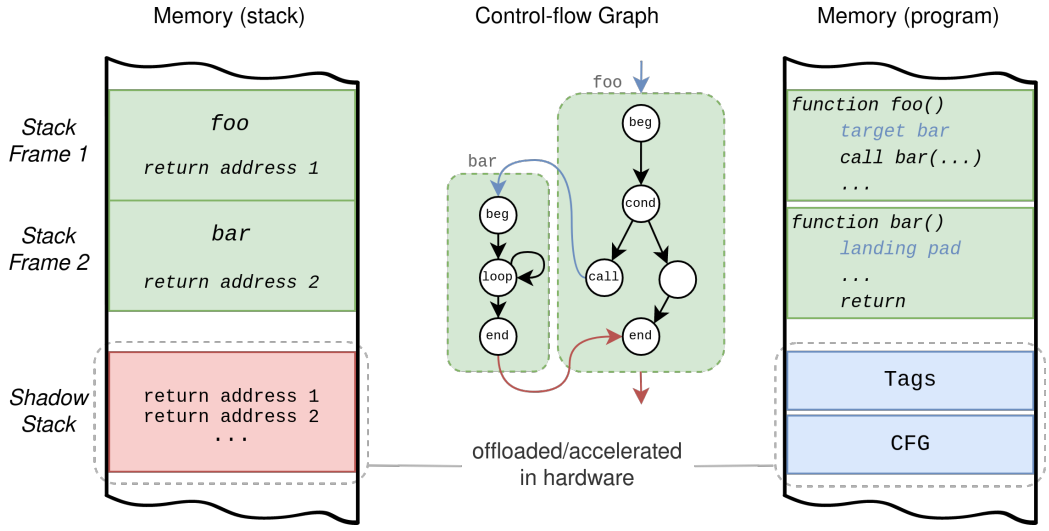


Figure 3.4 – Hardware CFI solutions, backward-edge (left) and forward-edge (right).

Forward-edge protection: For *forward-edge* CFI protection, landing pads define the only valid `branch/jump` targets. Intel defines Intel Indirect Branch Tracking (IBT) [21, Vol.1 Ch.17] and ARMv8 introduces Branch Target Identification (BTI) [25, Ch.B6] to add support for landing pads. The right part of Figure 3.4 presents examples of code instrumentation to support landing pads. For the RISC-V ISA, PHMon has been extended to define forward-edge CFI [157] in tandem with μCFI [139] for a RISC-V backend (it initially uses a run-time monitor built on top of Intel PT [21, Chapter 33]). μCFI extracts context information and derives Unique Code Targets (UCT) from the binary at compile time. The verification of the UCT is performed through three Match Units (MUs), one for indirect calls where the target is extracted and one for context data. FIXER also proposes a forward-edge policy that requires a static or run-time analysis of the program to extract the Control-Flow Graph (CFG). The CFG is stored as a 64 by 64 policy matrix in offloaded memory, where any check on an input caller should match the callee. The implementation is limited to 64 call-site addresses and requires preprocessing and analysis of the binary. Bratter [23] uses RISC-V "hints" to define instructions that store and

check branch tags in a dedicated CSR. It is limited to four concurrent tags in the CSR and requires the integration of additional instructions at compile time through binary scanning. In addition, a RISC-V specification [158] is currently being drafted to support both shadow stacks and landing pads. These solutions accelerate the CFG verification in dedicated hardware. Different levels of CFG precision are used, either forcing control-flow changes to function or basic-block entries or using a complete function CFG. All solutions require the binary to be compiled with additional information, a more precise solution impeding the compilation time.

3.4.2 Memory Tagging and Pointer Extension

Memory Tagging: To enforce temporal and spatial memory safety, which violations are among the most used attack vectors, ARM-v8.5 introduces the hardware support of Memory Tagging Extension (MTE) [24]. To tag code or data memory, a separate part of memory stores one-to-one tag association to data. The corresponding tag is embedded in the pointer-free space and checks at run time for a match. At compile-time, binaries are instrumented and associated "*colors*" to divide them into isolated regions [159]. The idea of embedded metadata to give pointer access to a region of memory is also extended with the use of "fat pointers" and associated capabilities that go past the tag itself. This concept is implemented by TIMBER-V [27] on the RISC-V ISA, which provides tagged memory to define isolated regions named "*enclaves*". It separates a `root` region, a supervisor region, and untrusted application regions. It provides a trust manager called TagRoot that bootstraps the enclaves and maintains isolated execution. While its memory overhead is low due to the reuse of shared memory among enclaves (through stack and heap interleaving as presented by the authors), the performance penalty of this isolation is averaged at around 25%.

Pointer Authentication: As landing pads only provide coarse-grained CFI, authentication and signing of memory pointers help refine it. A Pointer Authentication Code (PAC) is computed before a control-flow change using a lightweight algorithm. As the actual address space in 64-bit architectures uses less than 64 bits, the PAC is stored in the unused space. ARMv8 implements pointer authentication on indirect jumps through PAC [25, Ch.B6], and on return addresses through PAC-RET [25, Ch.B6]. It adds instructions to compute the PAC from the address itself and a context (*i.e.* values of the stack pointer or link register). Its main principle is presented in Figure 3.5. "PAC it up" [160] and PACStack [161] extend the principle to authenticate the call stacks through Message Au-

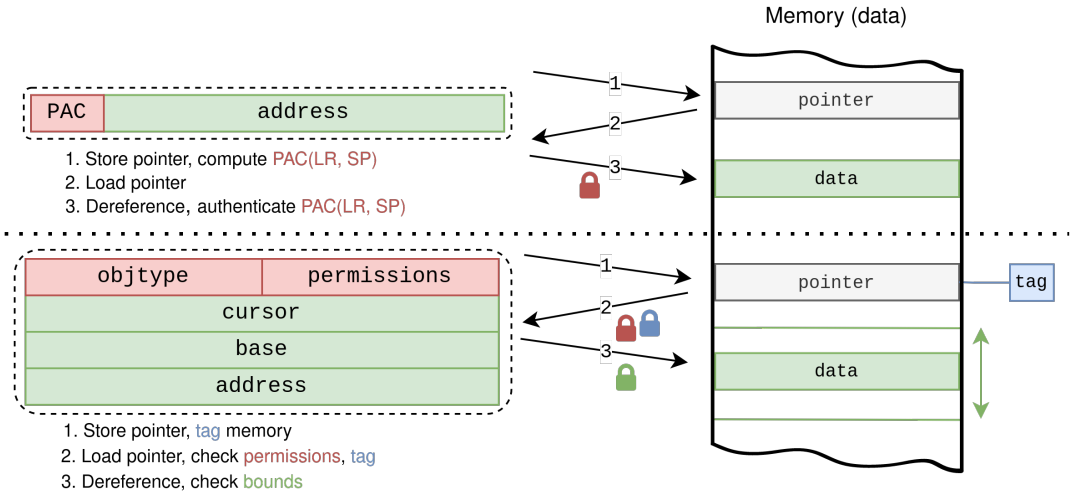


Figure 3.5 – Memory tagging and pointer extensions.

thentication Codes (MACs) using both the address and the MAC of the previous address to generate the newest one. On RISC-V architectures, RetTag [26] guarantees return address integrity through PAC authentication, and Zipper [162] chains the return addresses using MACs. These solutions all define integrity through pointer signing rather than duplication of the call stack. Calls and returns are instrumented with encryption/decryption of the signature and the performance overhead is directly linked to the hardware implementation of this mechanism. Similarly to the previous subsection, adding guarantees to the forward-edge flow transfers is notoriously more complex. For example, the base MTE usage only provides probabilistic checks, making it an offline software development tool. Extended with PACStack [160] to provide run-time safety, the performance overhead reaches 13%, and code-size overhead 22%. Overall, PAC and its derivates provide authenticity guarantees on the addresses used without the need for a dedicated shadow stack.

Capabilities: Extending the metadata addition in pointers through unused address space, even more metadata is appended to the address by defining "*fat pointers*". The fat pointers add several authentication values to the metadata of the pointer, generated at runtime, as presented in Figure 3.5. Among them Shakti-MS [163], a RISC-V processor extension that enforces memory safety adds a Stack Frame Cookie (SFC) unique to all functions, a RODataCookie (RODC) to protect the read-only segment of the program's memory, along with a base/bound couple that determines the maximum permissible range

the pointer can access. Along with the metadata embedded in each pointer, the authors define wrapper functions to generate and verify them through "*safe*" `malloc` `free` wrappers that generate and verify the fat pointers. Capability Hardware Enhanced RISC Instructions (CHERI) [29], [30] complement the fat pointers with both a base and offset to determine the maximum bounds of the memory access, along with permissions linked to the pointer (*i.e.* fetch instructions, load/store data, load/store capabilities, *etc.*). A hidden validity tag is associated with each physical memory location that can hold a capability, indicating the presence of a valid capability. Stores and loads are atomic with their corresponding tags, maintaining a one-to-one mapping safe even with concurrent accesses. CHERI first started as a prototype extension to the 64-bit MIPS ISA, but has since been extended to major compiler toolchains, and defined the specification for the ARMv8 or RISC-V ISAs [30]. These mechanisms provide strong security guarantees at a heavy cost in code-size overhead (CHERI fat pointers are 256-bits long)

3.4.3 Domain Isolation

Memory Protection Keys: As presented in the previous chapter (Section 3.2.3), Memory Protection Keys extend the PTE 10 unused bits to store permission keys associated with that page. They allow for a switch of domain and permissions while staying in the user space, making efficient permission switches. As an alternative to Intel MPK on the RISC-V ISA, SealPK [32] stores a key in the PTE space reserved for custom use and defines a co-processing unit that can hold up to 1024 keys, eliminating Intel's use-after-free vulnerability of the key, by tracking the page using each key at the OS level. Schrammel et al. define Domain Keys for in-process isolation (Donky) [31] building on the MPK support available in the Linux kernel. They provide a secure user-space software framework and monitor to protect domain permissions along with a lightweight hardware extension that stores the different domain policies through a dedicated CSR. They also implement a dedicated CPU exception triggered on incorrect accesses or modifications of the CSR. The (currently in draft) RISC-V N extension that implements user interrupts, is used to integrate a Donky hardware call gate to cross domains. It is also used to filter any system call from the user space [1]. The speed-up of such solutions to transfer memory permissions in a process is multiple orders of magnitude faster than regular context switches.

1. Note: instruction filtering often comes along in security monitors, preventing their usage from the untrusted region. FlexFilt [164] provides a co-processing unit dedicated to the setup and application of flexible instruction filters at run time.

These two solutions are presented on the left side of Figure 3.6, with their corresponding monitors.

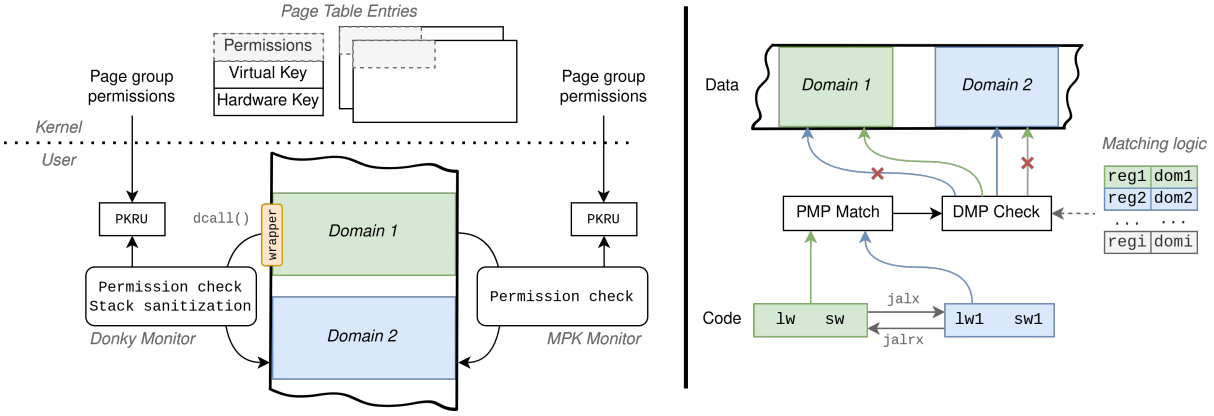


Figure 3.6 – Domain Isolation, PTE-based (left) and instruction-based (right).

Instruction-level Domain Isolation: Without the need for a monitor to define domains, dedicated instructions linked to each domain present similar in-process isolation at a lesser instrumentation cost. To this end, the RISC-V Physical Memory Protection (PMP) unit, which already defines different memory regions, could be extended with tag information. RIMI/DEMIX [33], [34] add a Domain Memory Protection (DMP) unit on top of it to attribute domain information to each region. The authors duplicate memory access instructions for each domain, tagging them with domain information, and verify that the current domain at execution matches the one extracted at the decode stage for a given instruction. It does not take precedence over the rules defined in the PMP and acts only as a complementary validation. The usage of the instructions themselves, as an isolation mechanism, provides a basis for a shadow stack, or domain isolation. Dedicated `load`/`store` instructions define the shadow stack (similar to 3.4.1) and its dedicated memory region set in the PMP/DMP. The mechanism can also be used to enforce domain isolation through dedicated domain switch instructions tied to specific domains, along with separated data access from each domain, once again with their dedicated memory access instructions. As an alternative to PTE-based domain isolation, this instruction-level design of domains provides a low overhead for strong isolation guarantees.

3.5 Summary

Recently, complex attacks against the RISC-V Instruction Set Architecture (ISA) have been conducted, and these are expected to become more frequent and intricate. Therefore, it is essential to anticipate potential future attacks to establish effective defenses. Similarly to other architectures, the shift to hardware-supported fine-grained run-time protections is notable in the RISC-V architecture. Hardware-based solutions benefit from the space left in RISC-V standards for extensibility at several levels. The implementation of custom instructions and their associated hardware support is multiple and opens up the space for prototyping in concrete use cases.

The cost of co-processing: As presented earlier, the RISC-V Rocket processor implements a coprocessor interface named RoCC that provides a unified way to build a hardware unit that defines custom instructions (as fixed in the RISC-V standards). Similarly, the CVA6 processor defines the CV-X-IF coprocessor interface, compatible between all cores of the CORE-V family (CVA5, CVA6 as application class cores, or CV32E40P/S as embedded class cores). These interfaces provide a way to prototype and implement custom instructions while keeping the main core untouched. They guarantee compatibility with future standard extensions by preserving custom instructions in their dedicated space. However, they restrict access to the internals of the processor that might be required for the setup of more complex solutions. The balance between performance and invasiveness is one axis of the design space. The adoption of the solution mainly depends on its impact on the core resource utilization, from which we define *"hardware requirements (HR)"*:

> `HR1:` The area, latency, and energy overhead on the hardware design should remain minimal, whether the implementation uses a co-processing unit or changes in the pipeline.

The cost of portability: On the other hand, invasive (even minimal) changes to the core bring out the most performance but break backward compatibility and might discourage wider adoption. This compatibility break is also present at the level of the added instructions. RISC-V "hints" are understood as `NOP`s in the case the core does not implement the corresponding action, however, the bit space is reduced as it has to be embedded in existing instructions. Another example is highlighted by the RISC-V CFI [158] draft specification. The use of the instructions in shadow stack mode falls back to

having no effect, keeping instrumented binaries portable. Using only the custom stack as the control stack breaks compatibility with cores that do not implement the custom instructions. Another axis opens up in the design space where portability and backward compatibility must balance performance. The main parameters that define the possible adoption of the solution are the impact on the code size and code performance, from which we define:

> **HR2:** The impact on the instrumented JIT code in terms of performance and code size should remain minimal for the solution to be integrated.

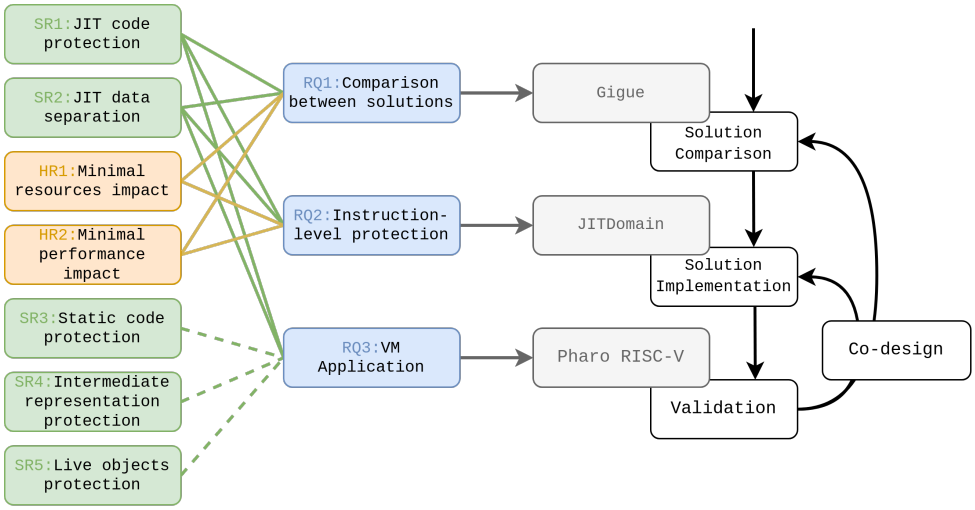


Figure 3.7 – Requirements, Research Questions and Contributions.

Both requirements define the design space axes and partially answer RQ1. They provide a basis to compare solutions between them in a common context, as this comparison is often complex and not investigated in the literature, comparison being made against vendor solutions or the corresponding software implementations. We extend the previously introduced figure with the hardware requirements in Figure 3.7.

Part II

Contributions

Chapter 4

Gigue: JIT Code Snapshot Generation for Hardware Testing

4.1 Motivation

As presented in the previous chapters, VMs are complex run-time environments responsible for the correct execution of a source language on a variety of architectures. Their widespread deployment and the impact of vulnerabilities discovered in them motivate the need for efficient and efficient secure solutions. The best results from the literature have been obtained using available hardware support for security primitives in commercial cores [9], [10], [165]. The development of such primitives in the open-source RISC-V ISA environment encourages prototyping. However, the design of hardware security primitives tailored for VM usage involves a large technology stack, ranging from the VM (or JIT compiler) itself down to the processor design under test. As it considerably slows down the development and implementation of solutions, we want to flatten the stack needed by a hardware developer, providing a meaningful but simplified workload. It would serve as a comparison of solutions and the implementations of requirements SR1 (JIT code protection), SR2 (JIT data separation), and investigate HR1/HR2 (impact on code size and performance) thus providing an answer to RQ1 (comparison between solutions).

4.1.1 Technology Stack

However, when designing custom instructions for the JIT compiler, a huge technology stack bridges the VM JIT compiler to the actual custom instruction support in hardware. Figure 4.1 presents the main elements involved. On top of the hardware, lives the operating system, providing base capabilities to the VM, itself distributed for the OS/architecture combination. Adding custom instructions would require the extension of the VM and OS along with the hardware itself (dotted circles on the figure).

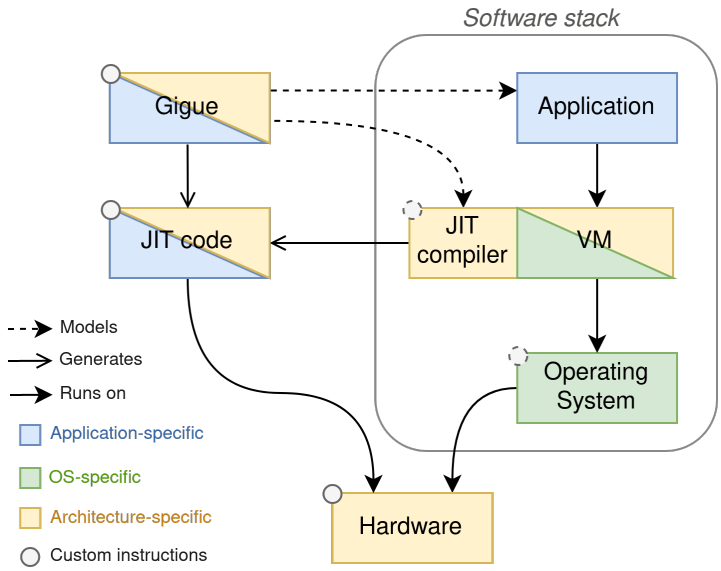


Figure 4.1 – JIT-oriented hardware development.

In the context of VMs, the technology stack ranges from high-level managed and dynamically typed source code down to generated machine code. Since the JIT compiler generates machine code at run time, we believe that it could use JIT-specific instructions dedicated either to common security measures (shadow stack [22], pointer integrity [28], or domain-based isolation [33]), acceleration measures (vector operations in the `V` extension [20], neural network operations [166]), or VM-specific measures (type checking, object offset verification). However, adding a new instruction to the JIT compiler requires support from both the OS and the underlying processor. As the RISC-V ecosystem is still developing, maintaining the whole stack covering all levels of execution makes it complex to prototype, test new instructions and their impact at the different levels of the stack. Alongside software complexity, the multiplicity of core implementations, either through their design or the extensions they support, makes it very complex to maintain or test ideas.

4.1.2 Existing Development Tools

Extended CPU emulators can be used to implement new hardware primitives and support their corresponding actions. Among them, QEMU [43], the Unicorn [40] wrapper on top of it, or gem5 [167]. Those emulators provide insight into both the executed software and its underlying microarchitecture. They can be extended to support new

instructions in two ways: addition to the simulated core itself, or instrumentation of "illegal instructions" exceptions. The binary itself then has to implement the custom instructions, either by generating the raw machine code bytes or extending an existing toolchain (*i.e.* GCC, LLVM).

On the hardware side, testing the correct implementation of the ISA in a CPU is two-fold: (1) ensuring the correct behavior of the ISA as defined in the standards, an assembly test suite (`riscv-tests` [168]) is defined to ensure the soundness of implementation; (2) ensuring the correct implementation of the components of the CPU, the different stages of the pipeline, memory hierarchy and peripherals through usage of "testbenches" and simulations. Soft-core implementations are written in Hardware Description Languages (HDL) that are compiled into "bitstreams" tailored to an FPGA model to deploy on a reconfigurable architecture for prototyping. Tools help simulate the design through waveform generation, among them closed-source commercial simulators such as Synopsys VCS, or Verilator [169].

To prototype new instruction ideas for the JIT compiler, we have to bridge the gap in the testing framework between the correct implementation in the JIT compilation pipeline, and guarantees enforced by the hardware itself. As stated in the introduction, the technology stack is complex, and portability is mostly guaranteed for all VM components (through major compilers) apart from the JIT compiler. Prototyping extensions on the hardware side could use a simplified version of the JIT code region and assess the impact on the core through four main metrics: (1) the performance overhead through the measured number of cycles, (2) the code size overhead on the instrumented code, (3) the area overhead of in FPGA resource utilization the solution adds, and (4) the impact on latency and the maximum frequency of the design. We believe having an insight into those metrics at the prototyping stage of a solution will guide our choice for implementation in the real JIT compiler. To ease prototyping, we designed Gigue, a tool that generates executable binaries similar to the JIT code region.

4.1.3 Custom Instruction Examples

To provide a better understanding of what Gigue aims to support, we present examples of custom instructions of increasing complexity that could be added to a processor. The first one consists of a hardware-accelerated primitive while the two others define hardware security primitives:

— **E1:** Dedicated instructions to handle bits rotation. These instructions are included in the RISC-V `B` extension that is not yet ratified and integrated into the standards.

— **E2:** A shadow-stack implementation with two instructions to push and pop a return address to a dedicated call stack, taken from the FIXER coprocessor implementation [22]. Any shadow stack pop is followed with a check for equality.

— **E3:** A pointer authentication of return addresses that signs addresses pushed on the stack and authenticates them on returns. It is built as a simplification of the ARM PAC-RET [25, Ch.B6] implementation.

The implementation of **(E1)** adds four instructions, two rotations between registers (`R`-type `ror` and `rol`) and the corresponding rotation using an immediate value (`I`-type `rori`). The implementation of **(E2)** adds two new instructions dedicated to the handling of return addresses during *calls* and *returns* that either push or pop it to the duplicated stack, using `spush` and `spop`. *Calls* should use the dedicated instructions for return address push, and *returns* should use the dedicated return address pop. **(E3)** is implemented by generating an additional instruction before calls, signing the address to which the control-flow is transferred. The signature is generated from the address itself and a context, the value of important registers. In this implementation, we used the link register (`ra`) and stack pointer (`sp`) along with a secret key. All calls are instrumented with authentication using the instruction `pac` and all returns are verified with the instruction `auth`.

4.2 Gigue: Design

To speed up hardware prototyping in the context of JIT compilation, we present Gigue, an open-source randomized workload generator that outputs executable snapshots similar to the JIT code memory region. It implements JIT custom instructions without the need to extend the whole technology stack to operate. As the JIT compiler [1] is the sole machine code generator, a focus on its output and how it can be instrumented provides good insight into the interest of one solution over another. Gigue is heavily parametrized to represent different JIT code regions (and therefore different VMs) as well as different application types running on top. Gigue usage is three-fold: (1) generate OS-independent JIT code regions that qualify different VMs or applications; (2) generate a set of instrumented binaries in `ELF` format, ready to be executed on a core; and (3) verify the execution of

1. Ahead-of-Time (AOT) compilers are not taken into consideration in this work as they generate code offline and can instrument it with precise CFI.

generated binaries using a CPU emulator (Unicorn [40]) extended on the software side to support the new instructions.

With Gigue, our objective is to ease the complexity of the technology stack involved in the support for custom instructions in the JIT compiler. Its design revolves around three main objectives:

1. *Parametrization:* Gigue is parametrizable to accurately qualify application classes. The parameters are covered in Section 4.2.2 and qualify both the size of JIT elements, the type of instructions they contain, interactions between elements and generated data.
2. *Modularity:* Gigue is modular and provides facilities to add user-defined structures that are found in the JIT code region (methods, PICs, hidden classes, *etc.*) and is presented in more detail in Section 4.2.4.
3. *Testing:* Gigue defines a test framework to guarantee the correct setup of JIT elements and the interpretation loop. It also checks the correct decoding and execution of the generated binary as presented in Section 4.2.5.

It generates a randomized executable workload similar to a JIT code region. The region structure was inspired by the Pharo VM JIT code region (see Chapter 6) but is adaptable to other architectures. The generated executable ELF file defines both an interpretation loop and a static version of the JIT code and JIT data. The JIT code elements (methods and optimization structures) are filled with random instructions based on input parameters.

4.2.1 Binary Structure and Execution

Structure:

The generated binary is composed of two main domains: a JIT code region that contains machine code structures usually found in VMs, and an interpreter loop that calls methods in the JIT code region in random order. Figure 4.2 presents the global structure of the binary, and presents the components of the JIT code region:

Methods: A method is characterized by its *size*, *call number*, *call depth*, number of *local variables*, and number of used *callee-saved registers*. It contains (1) a prologue that adds space on the stack to save callee-saved registers and set up local variables through the frame pointer; (2) a body, filled with random instructions and calls to other methods

or PICs; and (3) an epilogue, restoring register values saved on the stack and destroying the stack frame.

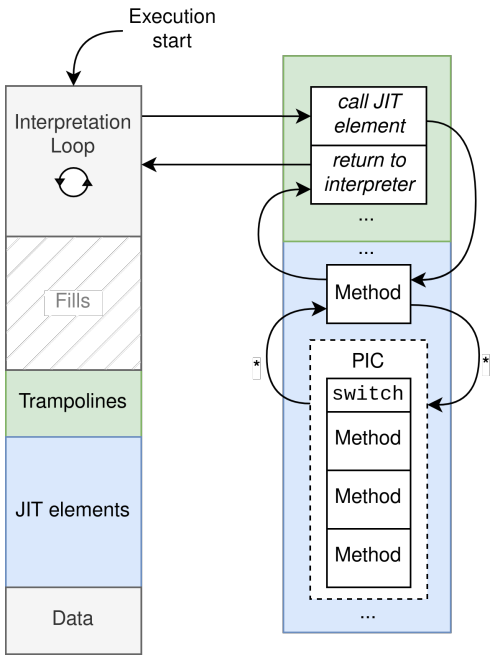
PICs: A polymorphic inline cache (PIC) is characterized by a *case number* and composed of a machine code switch case checking a corresponding class register for a value and jumping to the corresponding method offset. We use simplified class values (simple integers) and add the corresponding methods right after the "switch" statement. Calling a method in a PIC requires loading a corresponding value into the fictive class register before issuing a jump to the switch statement.

Trampolines: A trampoline is a machine code helper added to the JIT code region and used to handle the interoperability between the interpreter and the JIT code. Trampoline usage covers a wide variety of utilities, from accessing object field offsets to type-checking JIT methods and PICs, and switching execution stacks. The base generation Gigue provides simply defines control-flow trampolines to correctly transfer the control flow back and forth between the two parts of the binary: `callJITElt` and `returnToInterpreter`. The *call* trampoline is targeted by the interpreter calls. It stores the incoming return address on the JIT stack, changes the link register to the *return* trampoline and transfers the control-flow to the initially targeted JIT method. The *return* trampoline restores the interpreter return address and hands back the control-flow.

Execution:

Gigue generates an ELF binary that follows the execution design presented in Figure 4.2. The JIT code region contains methods and PICs filled with random instructions. The interpretation loop calls every one of the JIT elements in random order and each element can call other JIT elements. A call from the interpreter goes through the *call* trampoline that sets up the return address to the *return* trampoline and transfers the control-flow. Starred arrows set the PIC case number before performing the call to a PIC. An assembly template incorporates the interpreta-

Figure 4.2 – Binary Structure and Execution.



tion loop, the JIT code region (at a fixed
offset), and generated data whose base address is stored in a dedicated register (ensuring correct data accesses from stores and loads). The template is then linked according to the test framework of the `riscv-tests` official repository, defining the correct behavior of a core when confronted with different instructions and scenarios as defined in the standards. The binary is ready to be executed bare metal on a RISC-V core.

4.2.2 Parametrization

Gigue parameters are shown in Table 4.1 and split into two categories: *VM Characterization* tuning the layout of the JIT code region and *Application Characterization* tuning the JIT elements. The usage of these parameters is detailed in the next sections.

Type	Description	Name
VM	JIT code region size	$\boldsymbol{size_{JIT}}$
VM	Frequency of JIT elements	$\boldsymbol{weights_{elts}}$
VM	Usable registers	$\boldsymbol{regs}$
App	Total number of methods	$\boldsymbol{nb_{methods}}$
App	Method size variation	$\boldsymbol{\mu_{size}, \sigma_{size}}$
App	Call occupation in methods	$\boldsymbol{\mu_{calls}, \sigma_{calls}}$
App	Call intricacy and depth	$\boldsymbol{\lambda_{depth}}$
App	PIC case number	$\boldsymbol{\lambda_{PIC}}$
App	Frequency of instructions	$\boldsymbol{weights_{instrs}}$
App	Data characterization	$\boldsymbol{size_{data}, generator}$

Table 4.1 – List of Gigue input characterization parameters.

VM Characterization:

Parameters characterize the JIT code region for a specific VM by specifying a fixed JIT code region size. It also specifies the types and associated weights of the different JIT elements (methods and optimized structures) along with available registers, *i.e.* registers used by the VM itself that will not be used by the application. We believe that these values help distinguish VM JIT code regions from one another.

Application Characterization:

Parameters characterize an application through three main categories of parameters: method characterization (number, size and size heterogeneity), PIC characterization (PIC number distribution), instruction characterization (frequency of each type of instruction), and data characterization (size and layout). We show in Section 4.3 how different types of application classes can be defined using those parameters. Together, these parameters help define different categories of JIT code snapshots. For example, memory-intensive cases are generated by pushing the instruction distribution in favor of `loads/stores`. Intense call applications are generated by pushing the call occupation and depths of JIT methods.

4.2.3 Code Generation

Main Generation Routine:

Gigue is designed around a `Generator` object, responsible for the construct of the target binary according to the input parameters. It manages the JIT code region by adding the different elements, starting with the chosen trampolines, then with methods and PICs. The elements are then filled with random instructions using the `Builder` object, the sole emitter of machine code in the design. The main `Generator` phases are presented in order:

1. **Trampolines installation:** Trampolines are added at the start of the JIT code region, their address is kept for future reference.
2. **Elements addition:** JIT elements are added to the JIT binary according to their corresponding weights incrementally. They are filled with instructions at that time and given arbitrary call occupation and depth. Their addresses are stored for call graph resolution.
3. **Call patching:** Once all element addresses are known and stored, Gigue patches each method with the corresponding calls to other elements. The call occupation parameter is satisfied through the number of calls. The call depth is satisfied by targeting elements with a call depth decremented by one.
4. **Interpreter calls generation:** The interpretation loop is generated by adding a call to each JIT element in a randomized order. The Gigue binary correctly executes when it manages to run through all the interpretation loop calls.

5. **Data generation:** The corresponding JIT data sections are generated with a given size, following the JIT code in the address space. This space is usually exploited to store object tables, constant values that are too wide to be embedded into instructions as immediate values, or code pointers.
6. **ELF generation:** A linker script installs the binary and data in their corresponding ELF sections, setting up `U`-mode and other CSRs if needed. It then outputs the self-contained executable, ready to execute on a core.

Randomness:

Weighted Sampling: To determine used registers, instruction types, and JIT elements to generate, Gigue uses a list of weights associated with the elements. In the case of registers, it makes it possible to add pressure on frequently used registers. For example, temporary register `t0-t6` or the hardwired zero `x0` are frequently used by routines and they should expect a stronger usage. Available instructions are split by type: `R`-egister, `I`-mmediate, `U`-pper immediate, `J`-umps, `B`-ranches, `S`-tores, and `L`-oads. Note that while load instructions officially are `I`-type instructions, they were isolated to have a better separation between arithmetic instructions (`R` and `I` instructions) and memory access instructions (`L` and `S` instructions). As an example, it is possible to qualify memory-intensive applications using a higher frequency for memory access instruction types.

Distribution Laws: Gigue uses probability distributions parametrized by its inputs. These distributions can be parametrized but have been set to meaningful *normal distribution equivalents* by default. The method size variation alters the mean method size using a (positive or negative) percentage following a *truncated normal* distribution. The call occupation uses the same principle but derives the percentage of the method body that should be filled with calls. The call depth follows a *Poisson* distribution, providing shallow call stacks when $\boldsymbol{\lambda = 1}$ but more complex when used with $\boldsymbol{\lambda = 4}$ for example.

Sanity Checks:

Due to the random nature of the generated workload, several sanity checks are performed at generation time to guarantee the correct execution of the program. (1) The control of the registers that are used by random instructions guarantees no crucial register is overwritten breaking the calling convention. They are fixed at generation through the input parameter. (2) The call patching is performed once the methods are filled with

instructions and a leaf method is generated. Methods of a given call depth can only call methods of the call depth strictly inferior. This call graph construction method and the presence of a leaf method (no call depth, no callees) guarantees the complete execution of the program. Through the call patching, we also prevent recursive and mutual function calls to avoid infinite loops. (3) Jumps and branches are sanitized to stay in the method body and avoid landing in the middle of a call to prevent any disruption to the pre-established control flow. (4) Data accesses are random through the offset but are monitored using indirect accesses through a dedicated base register.

4.2.4 Modularity and Extensions

For VM characterization, each JIT element (trampolines, methods, and optimizations) answers a simple API to be added to the binary through the `Generator`. New constructs are added through a subclass of a JIT element along with its helper and probability weight. The only element that handles machine code instructions is the `Builder`, responsible for both outputting lone instructions and constructs such as calls or prologues/epilogues. It also enforces correct alignment for data access and sanitizes the landing of branch/jump instructions. Mechanisms using new instructions are integrated through a new subclass of the main builder, redefining the API of the main instructions and constructs it wishes to add or overload.

The three extension examples presented in Section 4.1.3, **E1** (rotations), **E2** (shadow stack), and **E3** (pointer authentication) were implemented in Gigue. Overall, the complete handling of the three different extensions requires the addition of the instructions encoding and a corresponding builder for changed constructs. The `Builder` element is subclassed to: add the new instructions to their corresponding types for **E1**, define different prologues/epilogues for **E2/E3**, and extend calls for **E2/E3**. Gigue is written in object-oriented Python code, the addition of **E1** took 161 lines, **E2** 154 lines, and **E3** 162 lines of code (counted using `cloc`, code formatted using `black`) respectively. These additions are mostly boilerplate code covering the instruction details and `Builder/Generator` extension. In addition to the code portion of the generator, and as presented in the next section, the test framework is also extended with a total of: **E1** 33, **E2** 31, and **E3** 53 to support the software execution of custom instructions in the test framework presented next.

4.2.5 Test framework

Gigue uses a test suite of around 150 parametrized unit tests (expanded to 2500) to assert guarantees on the generated binaries. Part of those tests directly run the generated machine code. To support the decoding and execution of machine code, we use the Capstone disassembler [170], another in-house disassembler for custom instruction unit testing, and the Unicorn CPU emulator [40]. It is a lightweight wrapper on top of QEMU that allows for simple binary instrumentation and tracing. We use Unicorn to execute the complete binary before generating the ELF file. It allows for quick tracing and smoke testing of the execution of custom instructions. It defuses early any issues that might be encountered when dealing with the real hardware.

```python
# tests/fixer/conftest.py
class FIXERHandler(Handler):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.shadow_stack = []

    def handle_cficall(self, uc_emul, pc, instr):
        fixer_reg = self.disasm.extract_rs1(instr)
        ra = uc_emul.reg_read(fixer_reg + 1)
        self.shadow_stack.append(return_address)

    def handle_cfiret(self, uc_emul, *args, **kwargs):
        ra = self.shadow_stack.pop()
        uc_emul.reg_write(UC_FIXER_CMP_REG, ra)
    ...
```

Figure 4.3 – Handler extension for the FIXER (**E2**) model.

In addition to simple baseline testing of unmodified binaries, we also adapted the Unicorn hooks to catch any unknown instruction. The custom instructions are emulated in the software directly before redirecting the control flow to the binary. We define a `Handler` base object that instruments the Unicorn execution by hooking several utilities if required. Among them, we define an *instruction tracer*, a *register tracer*, and an *exception tracer* for debugging purposes. In addition, we define a custom instruction handler API and hook: by defining a method named `handle_<name>`, the handler takes care of monitoring the PC, executing the corresponding software hook, and resuming the execution. Figure 4.3 presents how the `cficall` instruction from **(E2)** (simple shadow-stack) is handled when

the hook is triggered. A software shadow stack is used, and the value of the link register is read and appended to the list. On a return, the shadow stack pops the latest return address and places it in the dedicated register.

Any newly-defined handle function passed to a subclassed `Handler` like the one presented in Figure 4.3 is tied to a disassembler capable of decoding new instructions. To this end, we define `pytest` "fixtures", functions that set up and arrange the test before running the stimuli and assertion. A test function requests the result of a fixture by simply passing the fixture as an argument to the test. We define fixtures for all disassemblers, emulator setups, and handlers. Starting from the base `Disassembler` and `Handler`, we provide an easy way to compose the elements in the tests. In addition, a fixture at the scope of a function can activate the base Gigue logging for a given unit test.

4.3 Workload Qualification

4.3.1 VM Qualification

By the start of 2021, no major VM held a JIT compiler targeting the RISC-V architecture. We planned to extend the Pharo VM JIT compiler as a meaningful production VM that uses a simpler JIT compiler hierarchy than other high-level VMs. The Pharo VM uses a linear non-optimizing JIT compiler with registers fixed ahead of time. This extension is presented in more detail in Section 6. The predictability of the JIT compiler helps us extract meaningful information from its configuration parameters and associated JIT code region. To find corresponding VM parameters associated with VM description, as presented in Table 4.1, we gathered values from the Pharo VM implementation and previous analysis around the simulation environment used by OpenSmalltalkVM [38], from which the Pharo VM is derived. We also use information from the Pharo VM JIT code region raw machine code.

Code size: The size of the JIT code region is defined at startup by the JIT compiler and through an upper bound. In the case of the Pharo VM, the JIT code region is initialized with around 1MB of memory [2] (depending on the back-end) and an upper bound of 16MB [3]. This upper bound is set to allow inter-method calls and jumps to use

2. `https://github.com/pharo-project/pharo-vm/blob/pharo-10/smalltalksrc/VMMaker/CogAbstractInstruction.class.st#L1018-1026`

3. `https://github.com/pharo-project/pharo-vm/blob/pharo-10/smalltalksrc/VMMaker/Cogit.class.st#L9748-L9755`

small offset call and jump instructions if possible. Keeping jumps constrained reduces the number of instructions needed to redirect the control flow and avoids embedding sensible jump targets in the JIT data.

Registers: The used registers are defined and fixed ahead of time for their intended usage. All registers apart from the program counter, link register, stack pointer, and frame pointer, are passed as arguments into Gigue. A *class register* is used during the polymorphic inline cache switch phase to determine the correct receiver. Note that as we initially used all registers available, once we ported the Pharo JIT compiler to RISC-V (presented in Chapter 6), we retrofitted the list of registers to match the ones used by the VM.

	Number of sends	% of linked sends
Monomorphic	3566	90.4 %
Polymorphic	307	7.8 %
Megamorphic	70	1.8 %

Table 4.2 – Inline caches usage analysis [38].

Code elements: The frequency of the different JIT code elements (methods and PICs) is extracted from Miranda et al. [38] analysis of the JIT code region. They present the rich simulation framework used for the OpenSmalltalkVM development and extract information from the simulated JIT code region. They stop the simulation when the machine code zone reaches 1MB and present its contents. In Pharo, every "*call*" to another method is a "*message send*" to another object. When the JIT compiler first generates the corresponding machine code, it generates placeholder code to call the trampoline send routine - "unlinked sends" - that are then resolved and inlined along with a type guard when executed for the first time - "linked sends". Inline caches are either *monomorphic*, holding the call site they target, *polymorphic*, holding up to 6 different cases, or *megamorphic*, holding up to any number of cases. Their distribution is presented in Table 4.2, showing that 90.4% of linked sends (*i.e.* resolved call sites) are monomorphic.

4.3.2 Application Class Qualification

We define an application class in the JIT code region through the size of the methods, the call pressure between methods, the representation of different types of instructions, and the size of the associated data.

Call Pressure: Extracted from the same analysis using the simulated environment, information on the JIT code region is presented in Table 4.3. The authors show that 1752 methods are found in the JIT code region, and, as they stopped execution after reaching 1MB, 600 bytes long methods on average. However, this method size embeds both the metadata, type guard, prologue, epilogue, body and method-local data. Each method contains 3.63 sends on average, however, 37.9% of those sends are unlinked (*i.e.* never used), bringing down the used sends per method to 2.25 on average. To calculate the call occupation (percentage of the method body that performs calls), we use the method mean size, average call per method, and the call size, totaling 3%.

Number of methods	1752
Number of sends	6352
Average number of sends per method	3.63
Number of unlinked sends	2409
Percentage of unlinked sends	37.9%

Table 4.3 – JIT code region analysis [38].

Instruction distribution: The representation of the different types of instructions is two-fold, in their usage and their encodings. The instructions encodings are tied to the standards, and in the case of RISC-V, consists of: `I`-mmediate, `R`-egister, `J`-ump, `B`-ranch, and `S`-tore. Loads in RISC-V are encoded using the `I` encoding, which we define in Gigue as `L`-oad. As Gigue does not implement floating-point instructions, the link between the usage representation is direct: integer arithmetic (`I` and `R`), memory accesses (`L` and `S`), control-flow changes (`B` and `J`), and floating-point arithmetic (unimplemented). Cross-language benchmarks designed around VMs, such as "Are We Fast Yet?" [171], are used to compare language implementations and do not provide sufficient information on the underlying JIT machine code. The same issue occurs with benchmarks built around specific languages or platforms such as SPECjvm2008 [172] or DaCapo [173] around the Java VM, or Jetstream [174] and Kraken [175] around JavaScript.

	Arithmetic	**Memory Access**	**Control-Flow**
`min`	37.29%	6.57%	2.14%
`max`	79.43%	34.76%	41.86%
`geomean`	60.52%	18.87%	14.62%

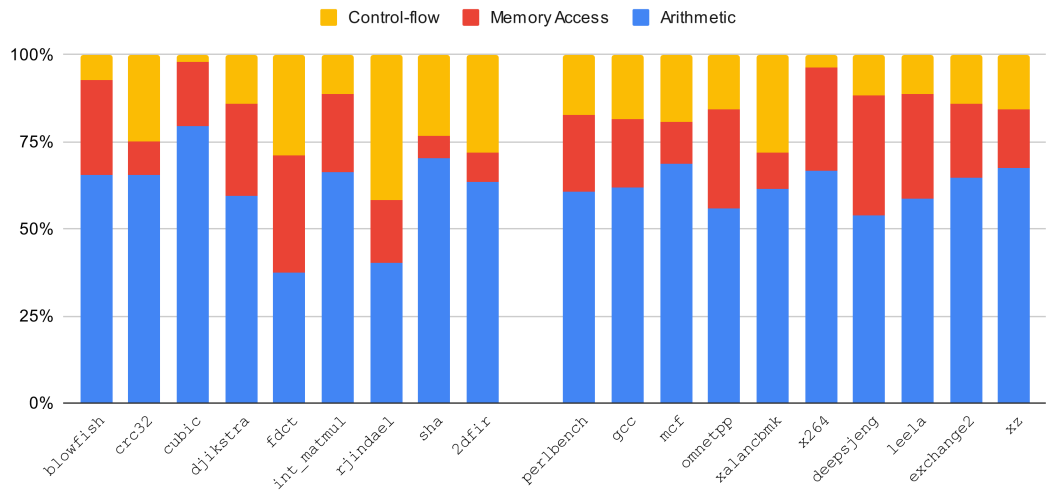


Figure 4.4 – Instruction execution distribution of the BEEBS [36] benchmark suite (left) and SPEC CPU-2017 [37] (right).

To define these representations, we use execution analysis of the SPEC CPU-2017 benchmarks [37] (Intel Core Xeon), and the BEEBS [36] (ARM Cortex-M0) benchmark suites. Those benchmarks are used to characterize CPUs and the energy consumption of embedded platforms. From the two referenced studies, we extract the instruction distribution and report them in the aforementioned categories in Figure 4.4. The corresponding ranges are displayed in Table 4.4 to present the available design space to explore.

4.3.3 Summary

We derive input Gigue values from the above metrics: the ratio of the size of the JIT code region and number of elements is kept (but minimized due to simulation times), the list of usable registers derived from the fixed registers the Pharo VM uses, and the weights of JIT elements (with an additional custom distribution for PIC/MIC distinction that reproduces the distribution presented earlier) in the Pharo VM. We also propose a baseline for application qualification by defining the number of methods and their call occupation, profiles for instruction type distribution extracted from existing benchmark analysis. From these parameters, we generate and augment application classes and workload.

```
"registers": [0, 5, 13, 14, 18, 19, 20, 22, 24, 25, 26],
"elts_weights": [90, 10],                    // [methods, PICs]
"pic_ditribution": "custom_pic",             // 8% PICs, 2% MICs
"jit_size": 20000,                           // Ratio kept for
"jit_nb_methods": 140,                       //   ~600 bytes methods
"call_occupation_mean": 0.03,                // ~2-3 calls/method
"instr_weights": [20, 20, 20, 12, 12, 8, 8], // [R, I, U, S, L, J, B]
"method_variation": [0.2, 0.1]               // (mean, standard deviation)
"call_occupation": [0.03, 0.03]              // (mean, standard deviation)
"call_depth_mean": 2,                        // Poisson parameter
"data_size": 2000,                           // Data size (1/10 JIT code)
"data_generation_strategy": "random",        // Generation algorithm
```

4.4 Use Case

Gigue provides a synthetic workload that uses custom instructions for a hardware developer. To this end, we present the capacities of Gigue by generating different application classes with varying call numbers and memory access intensities. We show how we qualify both VM and applications and two subprojects from Gigue that ease the deployment and running of benchmarks. Prelude handles the patching of the compiler toolchain and the generation of simple test samples implementing new instructions. Toccata handles the workload generation, running and results gathering on input cores.

4.4.1 Workload Generation

With input parameters defined, we augment the workload around two parameters: first the number of calls performed throughout a method, and second the number of memory accesses. In addition, we vary the number of methods and their corresponding sizes to amplify application examples besides the initial example. For calls, the initial setup uses a call occupation of 3% of the method bodies. We derive the call occupation to 1% and 6% of their method bodies to add new scenarios. Similarly, the base setup uses 8% of the instructions in the method bodies to access memory (through `load`/`store`). We derive the memory access intensities to 4% and 12% to add new scenarios.

We also experiment with the number of methods and their sizes by fixing the total

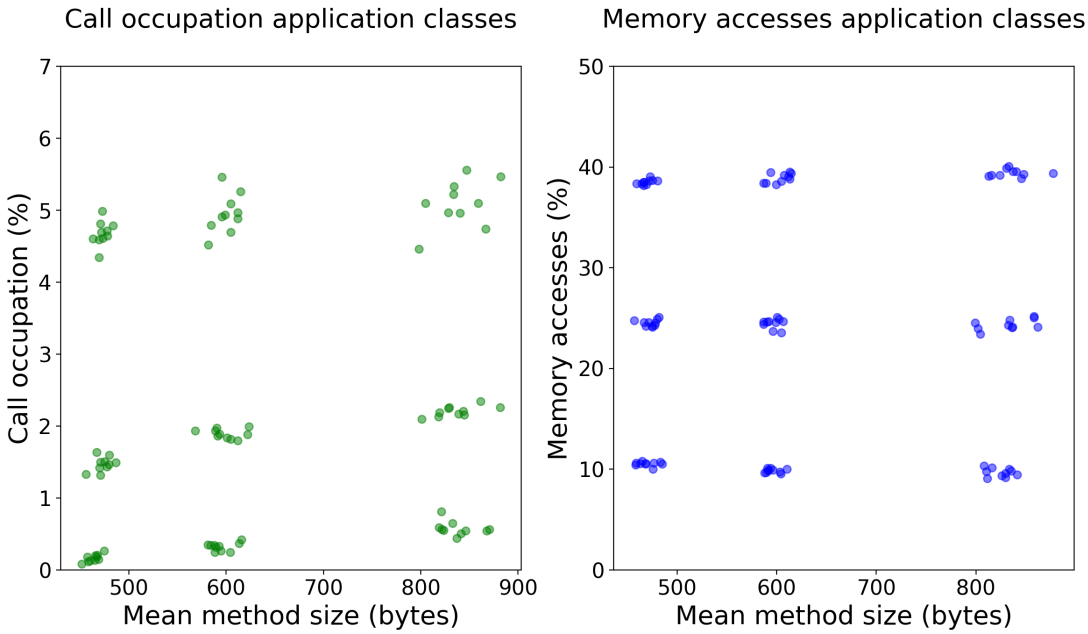


Figure 4.5 – Application classes generation.

size of the JIT code region, and augmenting the base number of methods that generates ∼600 bytes methods to also generate ∼400 bytes or ∼800 bytes methods. The three axes of exploration are presented in Figure 4.5, with the left part fixing the base memory access intensity and varying the call occupations and method sizes, resulting in 9 "application classes". Similarly, the right part fixes the call occupation and varies both memory access intensities and method sizes, defining 9 other "application classes". Note that the application classes are categorized here using their tracing results gathered by Toccata (presented in the next subsection) and contain both prologue/epilogue instructions, along with system utilities in the case of longer binaries, offsetting the input parameters. For example, the memory accesses contain the additional stack frame setup instructions, and calls are affected by prologue/epilogue instructions as well.

4.4.2 Experimental Setup

To execute the generated binaries, we use Toccata, a workload runner that instruments Gigue through varying input parameters, runs the generated binary on top of the Verilator model of a core, then collects and parses the execution results. We interface Toccata with two cores written in different Hardware Description Languages (HDL): Rocket [39], written in Chisel, a high-level HDL that extends Scala, and CVA6 [3], written in Sys-

temVerilog. The experimental setup is presented in Figure 4.6, both cores at the bottom along with their final objective of FPGA deployment through Vivado. The sources of both cores are used to generate their corresponding Verilator [169] models (cycle-accurate simulation). Files in green represent results that Toccata collects for each Gigue generation and run on top of a given core. Files in red represent custom instruction support, both in the compiler toolchain (for disassembly purposes), Gigue (for generation), and in the dedicated cores (for architectural support). To demonstrate Gigue generation capabilities, we use unmodified cores and the base ISA as presented earlier.

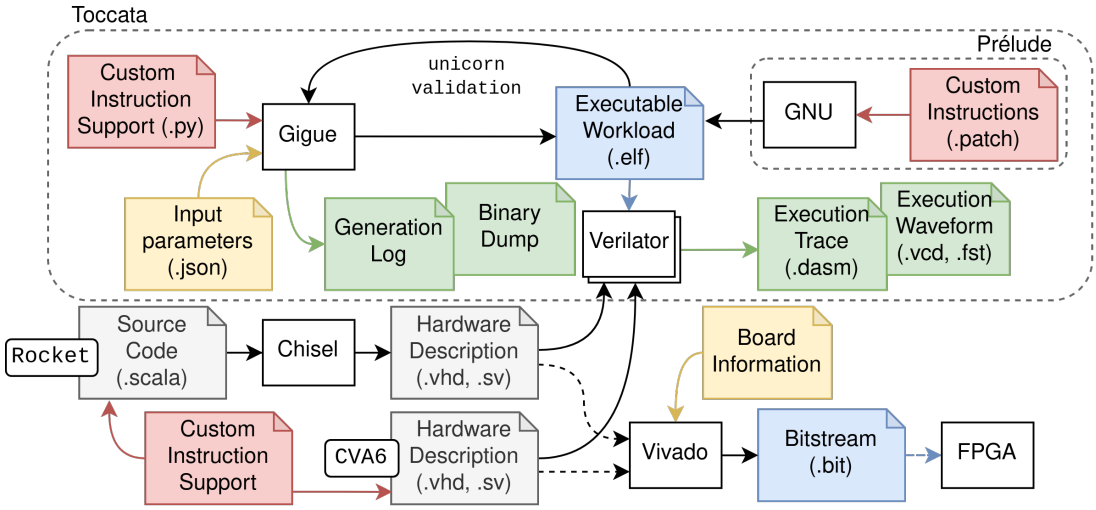


Figure 4.6 – Execution setup and test framework.

We ran the workloads under Ubuntu 20.04.6 LTS, on an Intel Core-i5 machine equipped with a 1.60GHz CPU and 16GB of memory. We fixed both core versions: version 1.6 for Rocket, and commit `bb80b3f` for CVA6 (in the absence of proper version tags). Both cores use Verilator version 5.008, and execute binaries linked and disassembled with the RISC-V GNU toolchain version 2.40.0. Figure 4.7 presents the execution of the workloads generated in the previous section on both cores.

Figure 4.7 presents the execution of the 9 call application classes and 9 memory access application classes on both CVA6 (blue) and Rocket (red). We see that we can directly augment target binaries with heavier call usage, increasing the workload on the core. Large methods with a heavy call occupation (and an increased call depth) generate long workloads on the cores. We also see that the increase in memory access adds overhead to the overall execution of the binary, as memory instructions are non-trivial compared to arithmetic instructions.

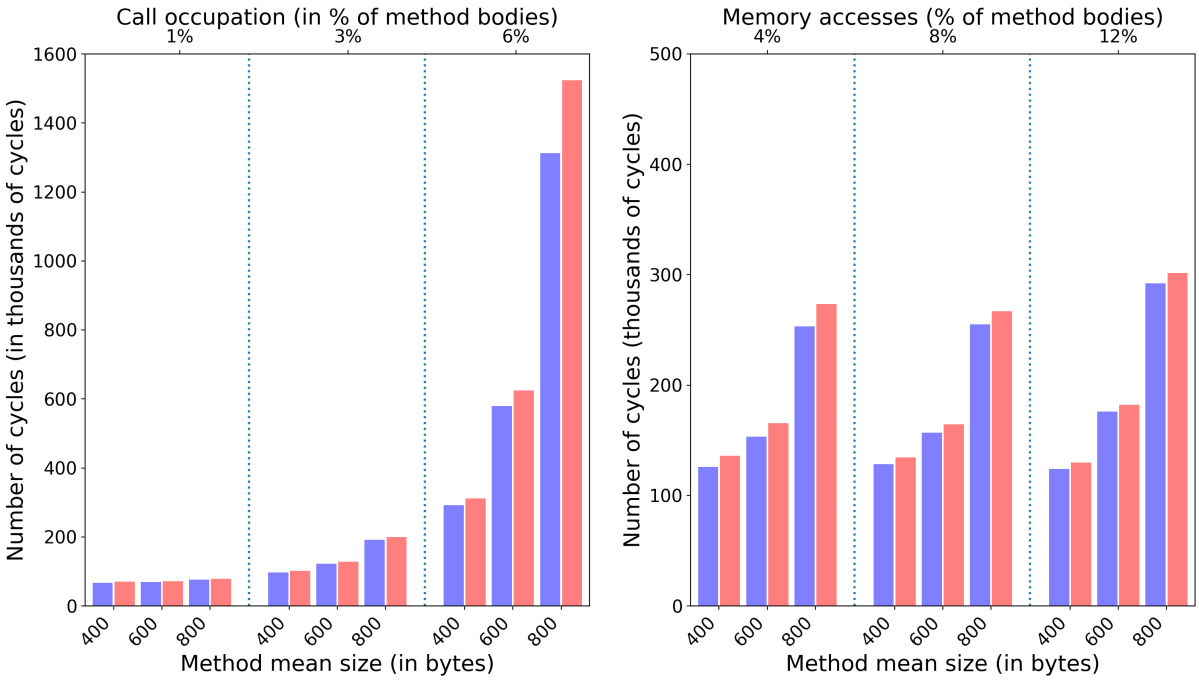


Figure 4.7 – Gigue workloads execution on CVA6 (blue) and Rocket (red).

4.5 Summary

In this chapter, we have presented Gigue, a modular and parametrizable random workload generator implementing custom instructions, and modeling the JIT code region. We have presented use cases of amplified application classes, with varying call occupations, and memory access intensities. Those application examples were derived from information on the Pharo VM and application benchmarks analysis. We presented the rich testing framework provided by Gigue to guarantee the complete execution of the generated binaries and show how it can be easily extended to support new custom instructions. We showed the direct interfacing of Gigue with different cores (Rocket and CVA6) through their Verilator model, using Toccata. Furthermore, the Gigue testing environment, and helpers provided by Prelude, help the hardware developer get minimal executable binaries implementing new instructions. To encourage reproducibility and usage, the Gigue code base is open-source and available on GitHub [4].

4. `https://github.com/QDucasse/gigue`

Chapter 5

JITDomain: Instruction-level security for the JIT compiler

Taking advantage of the JIT compilation process and locality of the JIT code, we present JITDomain, a hardware security framework based on instruction-level isolation [33], [34]. A minimally-invasive solution on an application class core such as the CVA6 [3] could later be compared with other solutions using a co-processing unit regarding their respective performances and security guarantees. We present a concrete implementation of defense requirements SR1 (JIT code protection) and SR2 (JIT data separation) to provide an answer to RQ2. We evaluate the solution on both its software impact through HR1 (impact on code size and performance) and hardware impact through HR2 (impact on resource utilization).

5.1 Threat Model

Attacker capabilities: We extend the threat model presented by JITScope [19] and extend it with more powerful attacks and up-to-date defenses [9], [10]. We assume the attackers have the following capacities: (1) attackers can write to any writable memory, and therefore corrupt control data such as jump/call targets and non-control-data such as bytecode, JIT intermediate representation, or internal structures; (2) attackers can read arbitrary mapped memory, and perform information leak attacks to bypass secret-based defenses such as ASLR. These assumptions are derived from common attacker capabilities and VM vulnerabilities defined in Chapter 2 through C1-C3.

We believe these assumptions are realistic as (1) memory corruption bugs are found each year in major VMs whether in the form of out-of-bounds, overflows, use-after-frees, or type confusion bugs; (2) *heap spraying* and JIT spraying [4] along with *heap feng-shui* [89] have helped defeat ASLR by injecting many objects in memory or the JIT code region to leak the address of an expected one.

Defenser capabilities: On the defense side, we assume (1) popular defenses such as ASLR or `W^X` are deployed by the operating system; (2) stack smashing mitigations are set up in the VM binary through compiler/preprocessor primitives, for example, canaries (through GCC or Clang `-fstack-protector`) and safer common functions (through GCC or Clang `FORTIFY_SOURCE`); (3) CFI measures are deployed on the static binary of the VM either using a hardware facility equivalent to Intel Control-flow Enforcement Technology (CET) [21, Vol.1 Ch.17] or ARMv8 Branch Target Identification (BTI) [25, Ch.B6], or a complete software version (through Clang `-fsanitize=cfi`). While the last requirements are examples from the x86 and AArch64 architectures, we believe similar solutions have been applied to RISC-V.

5.2 Software Execution Model

In this section, we present the framework built on assumptions on custom instructions that will be guaranteed through the hardware implementation and support for the solution.

5.2.1 Instruction-Level Domain Isolation

An instruction-level domain isolation [33] is set up by duplicating memory access instructions and assigning them domains of authorized execution. The code in a domain has access only to the corresponding data. Dedicated instructions are also added to change the current domain. Instruction-level domain isolation is based on three main principles, **(1)** the tagging of memory regions into "*domains*", **(2)** the duplication of memory access, and **(3)** the addition of control-flow transfer instructions to switch domains. Two checks are performed and presented in more detail in Section 5.3: **(1)** check that the instruction belongs to the domain it is being executed in, and **(2)** check that the instruction accesses data from a domain it has access to.

The instrumentation of dedicated domains for the JIT code region allows three main guarantees to be enforced on the JIT code: *call stack separation*, *data access restriction*, and *system call filtering*. To this end, we define three important domains: the *base domain* where the VM code and data reside (`basedom`), the *jit domain* that contains the JIT code and data (`jitdom`), and a *stack domain* that contains the shadow stack of return addresses (`stackdom`).

5.2.2 Memory Model

The memory model expected to support the framework is described in Figure 5.1 and is based on the Pharo VM memory model. On the left, low address, the native code of the VM is found in the `.text` region and contains the compiled code of the VM components: the interpreter, the JIT compiler, and the garbage collector. The next section `.data` (and other expected sections `.rodata`, ...) are the VM global variables. Those sections are the result of the compilation of the source code of the VM, OS-specific helpers, and dedicated plugins, linked together in an executable put in the *base domain*.

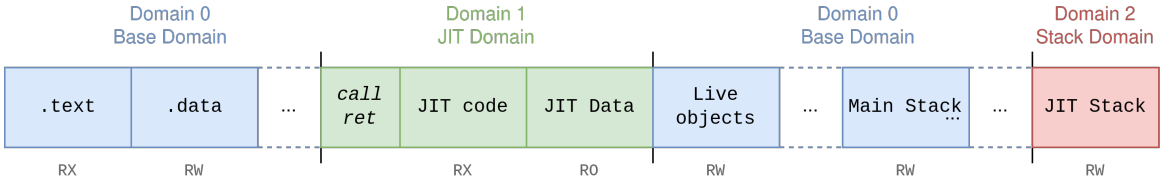


Figure 5.1 – Memory model of a VM implementing JITDomain.

The next section consists of the memory managed by the VM itself, in the heap. The executable JIT code region is allocated at startup and holds trampolines, the JIT code itself, and JIT data (code pointers, jump tables, ...). The `call` trampoline serves as a landing pad for interpreter calls, and the `return` trampoline that directs control-flow back are set up in the *jit domain*. Alongside, both JIT code and data are installed in the *jit domain*.

Live objects are also found in the heap, where they are placed and garbage-collected. In the case of a generational garbage collector, space for young and old objects is allocated. They are put in the *base domain*. Finally, at very high addresses, the main stack is found, that contains both local variables, function call frames, and return addresses. A separate stack used by the JIT code is allocated at the VM startup and managed differently. The main stack is put in the *base domain* while the JIT stack is set up in the *stack domain*.

5.2.3 Call Stack Separation

Motivation: The motivation for call stack separation of the JIT code is to set up a backward-edge control-flow integrity solution. Deploying CFI on JIT code is required to prevent the redirection of the control flow necessary to launch both code-injection [4] and code-reuse [5], [6] attacks. The dynamic constraint of JIT compilation forces usage of degraded versions of CFI for the JIT code [18], [19]. Isolating the complete JIT call

stack in a dedicated domain and memory region hides it from a potential attacker and exposes less information to the general call stack and interpreter.

Instrumentation: The RISC-V calling convention passes arguments in registers when possible (up to eight registers `a0-a7`). If the called function uses any of the *callee-saved* registers (`sp` and `s0-s11`), it has to store them on the stack. In addition to those, if the called function performs calls on its side, the return address `ra` has to be pushed on the stack. Spilling control-flow information to the stack opens the door to potential attackers.

```
jit_method:
    # Prologue
    addi sp, sp, -16
    sw   s0, 0 (sp)
    ...
    # Shadow stack save
    addi ssp, ssp, -4
    sst  ra, 0 (ssp)
    ...
    # Shadow stack load
    lst  ra, 0 (ssp)
    addi ssp, ssp, 4
    ...
    # Epilogue
    lw   s0, 0 (sp)
    addi sp, sp, 16
    ret
```

Figure 5.2 – Call stack Separation.

To isolate the JIT call stack, the prologue and epilogue of each JIT-compiled method are instrumented to use dedicated stores and loads, `sst` and `lst`. Those instructions are linked to the *jit domain* and are the only access point to the *stack domain*, the region where the shadow stack of return addresses is stored. We use it here as the only call stack the JIT code uses and as shown in the code snippet in Figure 5.2. This decision is motivated by the recommendations from Burow et al. [115] for performance and conciseness: the check for correctness is redundant as the correct shadow return address is loaded.

Discussion: A duplicated stack can be used in two ways, either to duplicate the existing call stack and perform comparison checks on function returns, in *"shadow stack"* mode, or operate as the only call stack, separated from the common stack and without performing comparison checks, in *"control stack"* mode. The RISC-V `Zisslpcfi` [1] extension draft presents both implementations and argues that the choice of one of the other is motivated either by conciseness for the *"control stack"* mode or portability for the *"shadow stack"* (where load/stores to the duplicated stack would boil down to `NOP`s on a system that does not implement the extension).

1. `https://github.com/riscv/riscv-cfi`

5.2.4 Data Access Control

Motivation: JIT data is set up in memory and accessed by JIT code. It consists of sensitive information such as code pointers, jump target addresses, or relocation tables. To guarantee its confidentiality and restrict access to the JIT data, only the JIT code should be able to access it.

Instrumentation: To separate the JIT code region from the rest, JIT code and JIT data are put in their dedicated *jit domain*. All memory accesses from the JIT code use duplicated instructions that **(1)** are only usable from this domain, and **(2)** can only access data set up in the *jit domain*. At run time, **(1)** a first hardware check is performed to ensure that the current instruction is executing in its dedicated domain, and **(2)** a second one to ensure it accesses data from the correct domain. More details on the checks performed are presented in section 5.3.

All duplicated instructions (*i.e.* ending with `_1`) are only executable from within the JIT code region and can only access JIT data. In contrast, the base memory accesses can be executed from both domains, as they set up the calling convention on the main stack. The logic to change the domain is handled by the dedicated instructions that duplicate `jalr`. The first one is used by the interpreter to transfer the control-flow to JIT methods, `chdom`, storing the target address and using a *"call"* trampoline. In response, `retdom` is used to transfer execution back from the JIT region to the interpreter from a dedicated *"return"* trampoline.

```
interpreter_loop:
    ...
    sw   t0, 0(sp)
    # To call trampoline
    la   t1, jit_method
    chdom call_trampoline
    lw   t0, 0(sp)
    ...

call_trampoline:
    ...
    j    t1
```

```
jit_method:
    # Loading JIT data
    lw1   t0, 24(s0)
    # Storing JIT data
    sw1   t0, 24(s0)
    ...
    # To return trampoline
    ret ra, 0(ra)

return_trampoline:
    ...
    # To interpreter
    retdom
```

Figure 5.3 – Instruction-level data isolation and domain transfers.

Discussion: Separating code and data is crucial both to guarantee data confidentiality

by restricting its accesses and to grant them different permissions. Authors of NoJITsu [9] define the JIT code as executable only and the JIT data as read-only. They then monitor the functions that install, patch, and delete JIT-compiled code with additional permissions.

5.2.5 System Call Filtering

Motivation: Powerful code attacks use one or several system calls as their final step to allocate memory executable memory at a known address. JIT code, either genuine, sprayed, or reused, contains user input and should not execute any powerful system call, even in its hidden execution path.

Instrumentation: To filter out system calls from executing in the JIT code region, we assign the RISC-V environment call (`ecall`) instructions to *base domain*. This way, these instructions cannot be executed from within the JIT code region in U-mode. Their availability is maintained outside that region.

Discussion: Preventing the system call instruction from executing during its decoding phase ensures the final step of a JIT attack cannot be performed from within the JIT code region itself and blocks the execution of injected or reused system calls. This filter also guarantees that even if an `ecall` instruction is present in the hidden execution path of the binary, it will be monitored and filtered.

5.3 Hardware Extension Design

The CVA6 CPU is a 6-stage, single-issue RISC-V core that implements the 64-bit version of the ISA [3] with extensions RV64IMAC. It implements three privilege levels and can support a fully-featured OS such as Linux through its support for memory virtualization (through the Memory Management Unit - MMU - and Page Table Walker - PTW -) along with branch prediction units. It is maintained by the OpenHardware group and the French defense company Thales. An overview of the CVA6 frontend, backend, and cache subsystem is presented in Figure 5.4.

We briefly present the six stages of the CVA6 and their main components. **(1)** The PC Generation (PCGen) stage; the next program counter is chosen from the different possible sources. Among them are the result of branch prediction, a trap vector if an exception is raised, or a simple increment. **(2)** The Instruction Fetch (IF) stage; from the

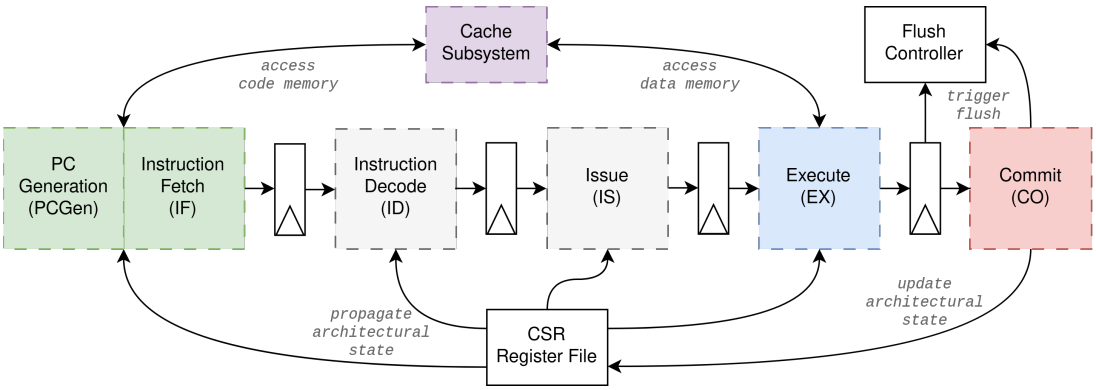


Figure 5.4 – CVA6 pipeline overview.

deduced next program counter, instructions are fetched from program memory, realigned and passed to the next stage. **(3)** The Instruction Decode (ID) stage; decodes the raw instruction input, formats them into a scoreboard entry and passes them to the next stage. **(4)** Issue (IS), the formatted instruction is attributed to a corresponding functional unit and placed into the scoreboard. Its status is traced, and this stage handles any write-back from the execute stage. **(5)** The Execute (EX) stage; contains all functional units that compute the result of instructions. It handles all arithmetic operations, branches, and memory operations through the MMU. Finally, **(6)** The Commit (CO) stage propagates any architectural state change as a side effect of executed instructions. It also provides an interface to the flush controller, responsible for complete or partial pipeline/memory flushes.

5.3.1 Instruction Tagging

The CVA6 core uses a scoreboard structure to keep track of decoded instructions and their operands, along with any result or exception. We extend this structure with two main parameters: a *target domain*, the domain that the instruction accesses either through memory accesses or control-flow changes, and a *domain change* flag that defines if the instruction should change domain or not. The two fields are added to the scoreboard entries and filled in during the decoding/issue phase. In addition to those two fields, the *code domain*, the domain the instruction should be executed in, is checked in the decoder directly and does not need to be propagated further in the pipeline.

The values of these three fields (*code domain*, *target domain*, and *domain change*) attributed to the different instructions, both default ones and duplicated, are presented

Name	Code Domain	Target Domain	Domain Change
chdom	basedom	jitdom	Y
retdom	jitdom	basedom	Y
b*	domi	*current domain*	N
j*	domi	*current domain*	N
l*	domi	basedom	N
s*	domi	basedom	N
l*1	jitdom	jitdom	N
s*1	jitdom	jitdom	N
lst	jitdom	stackdom	N
sst	jitdom	stackdom	N
ecall	basedom	-	N
default	domi	domi	N

Table 5.1 – Instruction Domain Tagging

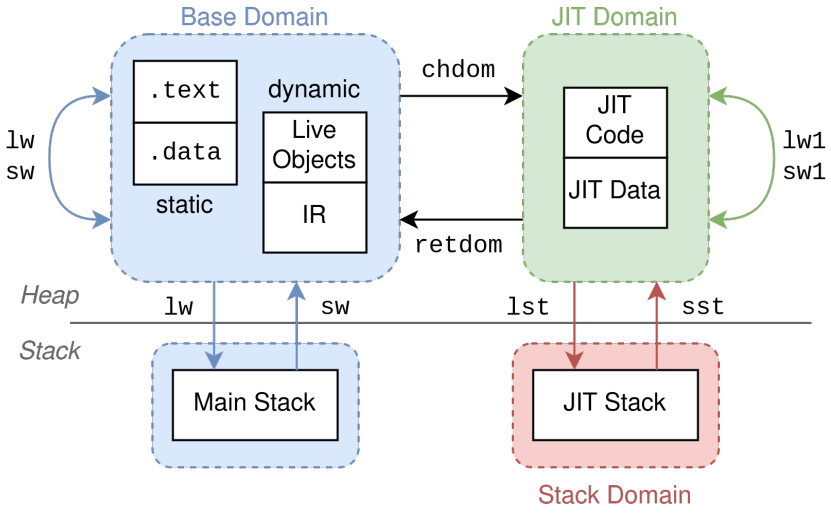


Figure 5.5 – JITDomain overview, instructions and their domains.

in Table 5.1. We use `domi` as a "domain-inclusive" tag, meaning it can execute from, or target both the *base* and *jit domains*. By default, instructions are tagged with the pair `domi/domi` as code and target domain, and the domain change flag is disabled. From the table, we can see duplicated memory accesses are restricted to the *jit domain* for their code and data accesses. Base memory accesses are executable from any domain, but can only access data in the *base domain*. The domain change instructions `chdom/retdom` are the only ones responsible for transferring between *base* and *jit domains*, `chdom` as the forward change and `retdom` as the backward change. They are duplicated versions of the jump-and-link-register (`jalr`) instruction. Next, all base control-flow changes can be executed from any domain but should not cross domains, and require a dynamic input for their target domain, corresponding to the current domain found at the decoding time of the instruction. Finally, the user-space environment call instruction is set to only execute in the *base domain*.

5.3.2 Control and Status Registers

Control and Status Registers (CSR) are registers that keep information on the internal state of the CPU. They can be read-only or modified through dedicated instructions. To implement the domain tagging of memory regions, additional CSRs are needed to store the domain information linked to regions defined in the Physical Memory Protection unit. The CVA6 core implements 16 PMP regions through the 64-bit wide `pmpcfg0` register, which contains configurations for PMP regions 0 to 7 (`pmp0cfg` to `pmp7cfg`), and `pmpcfg1` for regions 8 to 15.

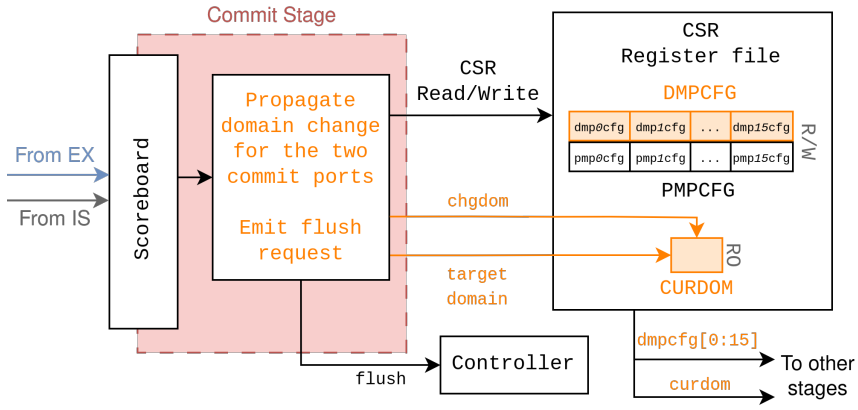


Figure 5.6 – Domain configuration and current domain CSRs.

Aside from this PMP configuration, we define a `dmpcfg` CSR that contains 16 domain configurations (`dmp0cfg` to `dmp15cfg`), displayed in the CSR register file in Figure 5.6. Each domain configuration is associated with a PMP memory region and contains a 2-bit field responsible for storing the associated domain and a locked bit. The configuration is 4-bit wide to keep space for additional domains if needed. We extend the PMP module with an additional expected domain as input and perform the domain check with the associated domain on the matched PMP region conservatively. The `dmpcfg` CSR is given an address in the `0x7C0-0x7FF` range to respect the "custom read/write" accessibility set in the standards [56]. The `DMP` is accessible through the usual CSR read/write privileged instructions `csrr/csrw`. The `dmpicfg` configurations contain a lock `L` bit that is functionally identical to the one present in `pmpicfg`. The activation of this field extends the DMP verification to M-mode accesses. This means that `dmpcfg` registers (similarly to PMP registers) are locked and cannot change until the hardware thread (hart) is reset.

We also store the current domain in a CSR named `curdom` displayed in Figure 5.6. The output from the CSR register file is directed to the other stages of the pipeline. During the commit of an instruction that has the domain change flag `chgdom` of its scoreboard entry activated, `curdom` is replaced with the target domain associated with the instruction, and the pipeline is flushed. As this register should not be modified from outside the core, it is given an address in the `0xFC0-0xFFF` range to respect the "custom read-only" accessibility [56]. The CVA6 uses two commit ports, the domain change is propagated when one of the two ports expects to change the current domain and the other does not affect it.

5.3.3 Domain Check: Code Domain

The first check is performed on the code domain through the decoder in the decode stage of the pipeline. According to Table 5.1 presented earlier, the decoder associates to all instructions a code domain, a target domain, and a flag for domain change that is passed to the issue stage in the filled scoreboard entry. The code domain associated with the instruction decoded instruction is checked against the current domain input, coming from the `curdom` CSR presented earlier. Any mismatch raises an illegal instruction fault, propagated in the pipeline. These modifications are presented in Figure 5.7.

In addition to the tagging of all instructions with new fields for code and target domains, the decoder is also extended to support the new instructions that duplicate memory accesses for both the *jit* and *stack domains* along with domain changes. Apart from the different domains, the duplicated `loads/stores` use the same data flow as the

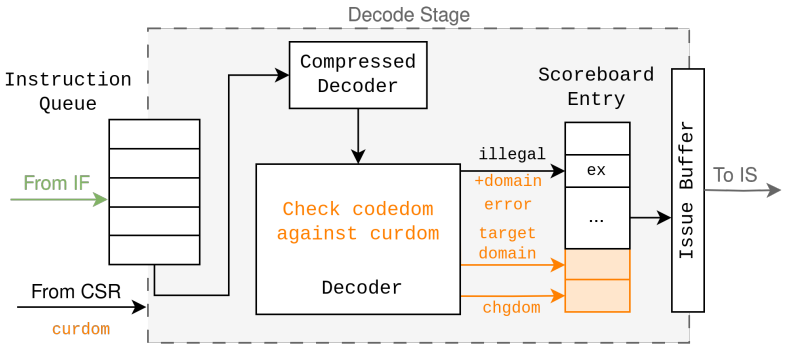


Figure 5.7 – Code domain check.

base ones. Similarly, `chdom` and `retdom` are modelled after `jalr`. The support for the duplicated instructions is presented in more detail during data access in Figure 5.3 for loads/stores, and instruction fetch in Figure 5.9 for `chdom`/`retdom`.

5.3.4 Domain Check: Data Domain

The load-store unit is one of the functional units embedded in the execute stage. It has to manage the interface to data memory, data caches, the hardware Page Table Walker (PTW), and the Memory Management Unit (MMU). For data accesses using stores and loads, the PMP is used on the final translated physical address, or at any point during the page table walk process.

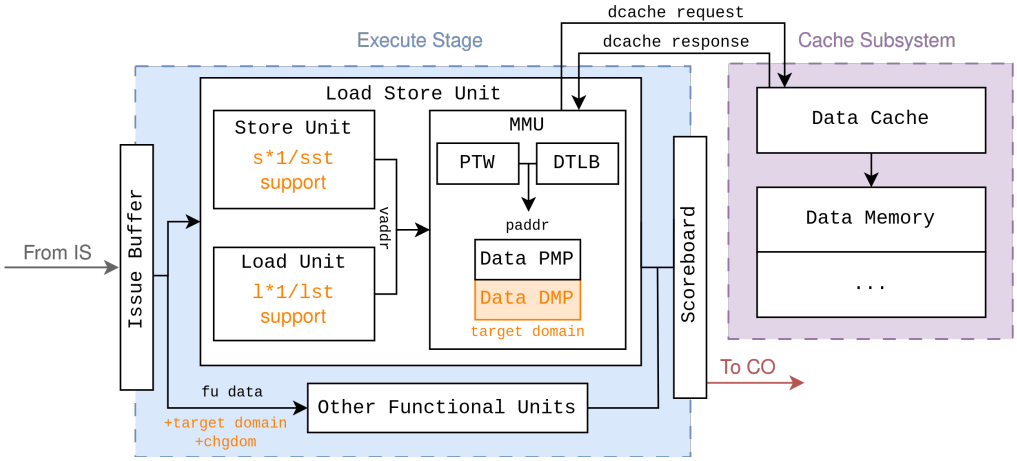


Figure 5.8 – Data access domain check.

As presented in Figure 5.8, we instrument the final PMP physical access with the expected target domain through the DMP. This expected target domain is passed along

in the functional unit data, from the issue to the execute stage. The DMP validates the access on top of the PMP, conservatively. In addition to the target domain check for data accesses, we also add support for the duplicated `loads` and `stores` in the Load Store Unit (LSU) when the decoded address is passed to the MMU.

5.3.5 Domain Check: Fetch Domain

The same idea is applied to the instruction fetch stage. The fetched address depends on the next program counter, deduced either through a simple increment from the previous one, branch prediction, or the resolved address obtained back from the branch unit after a mispredict. The instruction fetch performs a request to the cache, that in turn performs a translation request to the MMU on a miss. The different cache requests and responses are extended with domain information that is eventually passed to the instruction PMP and DMP in the MMU.

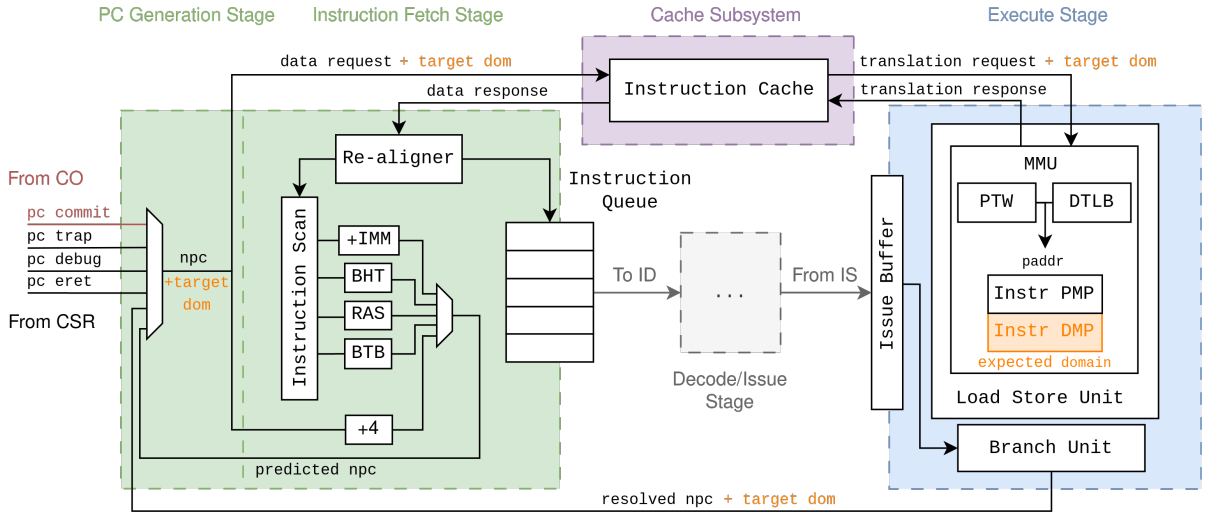


Figure 5.9 – Instruction fetch domain check.

The front end is conceptually split into two phases: the generation of the next PC (PCGen) and the instruction fetch (IF). The fetch request is given an expected target domain that depends on the next PC chosen. We distinguish three main cases: a base control-flow change, a domain change, and a higher-level exception or debug request. The next PC can originate from different sources (sorted from lower precedence to higher precedence):

1. **Default Assignment & Branch Predict**: By default, the PC is incremented to

PC+4, going to the next 32-bit word. Note that even in the case of compressed (`C`) instructions, the complete word is fetched. Alternatively, when the branch prediction is valid and passed, it depends on the different modules the front end implements, such as a Branch History Table (BHT), a Branch Target Buffer (BTB), and a Return Address Stack (RAS) to predict the next expected address to fetch.

2. **Control-flow change request**: When the above step fails and the branch predictor mispredicts the next fetch address, the fetch restarts from the resolved address obtained back from the execute stage. The branch unit charged to return the address extracted from the actual instruction also delivers expected domain information.
3. **Return from `ecall` & Pipeline flush**: A return from an environment call, or the way to perform a privileged call from a less privileged level, restores the saved PC from before the call. When CSRs with side effects are modified, the pipeline is flushed and the fetch is started over from the PC of the last instruction committed.
4. **Exception/Interrupt**: When exceptions or interrupts are triggered, the next PC is set to the trap vector base address, depending on the privilege level (handled by the CSR unit).
5. **Debug**: In addition, the debug unit can interrupt any control-flow requests, and it reports requests to change the PC directly to the CPU.

Those five families of PC generation are instrumented with expected domain information to correctly fetch the next instructions. As a reminder, only dedicated domain change instructions `chdom`/`retdom` are expected to fetch from a different domain than the current one.

Since we did not implement branch prediction for our custom instructions, the expected domain passed to the fetch request is set to `curdom` for the **(1)** default increment and branch prediction case. This current domain is also expected to be maintained throughout **(3)** `ecall` returns as they cannot be executed from the *jit domain* and should recover in the *base domain*. In addition, **(3)** pipeline flushes should keep the current domain as the effective domain is only changed at instruction commit time.

When a mispredict occurs **(2)**, the expected domain is obtained from the resolved address, back from the branch unit in the execute stage. The expected domain is then set to the `target domain` passed in the scoreboard entry. It should either correspond to the current domain, for base control flow changes, or the new domains for `chdom`/`retdom`.

In the case of an exception or interrupt **(4)**, we set the expected domain to the *base domain* to reach the trap vector base address set in it. The trap handler should restore

the current domain to *base domain* if the exception occurred in the *jit domain*. Finally, in the case of a debug request **(5)**, as the debug primitives require additional freedom, the expected domain is set to `domi`.

5.4 Evaluation

In this section, we present the evaluation of the implementation through the functional verification through a dedicated unit test suite, we evaluate the performance overhead of the solution using Gigue-generated binaries that implement the solution. We also present its FPGA resource utilization to demonstrate the hardware overhead of JITDomain.

5.4.1 Functional Verification

To guarantee the correct setup of the different domain checks, the PMP/DMP logic and update, instruction tagging and assert that the modifications of the design do not alter its execution, we developed an assembly test suite and runner that exhaustively covers the possible scenarios. The testing suite contains 134 unit tests split into six categories presented in Table 5.2. Those tests cover the main *code domain*/*target domain* checks for all memory access instructions, either from the base set of instructions, the duplicated ones used in the *jit domain*, or shadow stack accesses in the *stack domain*. The usage of PMP/DMP CSRs is verified using all combinations. Domain change instructions are checked according to their corresponding *code domain*/*target domain*, and similarly, base control-flow change instructions are tested to stay in the current domain. Pipeline flushes are tested and expected for both `chdom` and `retdom` to ensure the domain change is enforced for the next instruction. Finally, the prohibition of system calls is checked in the *jit domain*.

Category	**Number of tests**
Memory access instructions (base, duplicated and shadow stack)	48 + 57 + 12
PMP/DMP CSR integration and usage	4
Base control-flow changes for non-domain-changing instructions	3
Domain change instructions	6
Pipeline flushes on domain changes	2
System calls	2

Table 5.2 – JITDomain Validation Tests

Along with the assembly tests, we provide a suite runner that interfaces with the Verilator model of the core, gathers the results and reports them. We also provide a patch that integrates our custom instructions in the GNU compiler toolchain and the Spike emulator, to provide both a way to compile our simple tests, and disassemble binaries and log traces with our custom instructions. We execute the tests using the Verilator model and collect the trace results for validation. Our implementation of the JITDomain framework validates all tests from the suite.

In addition to the newly developed test suite, we also ran two regression test suites provided by the RISC-V foundation, `riscv-tests` [2] and `riscv-arch-test` [3]. The first suite defines 396 unit tests based on *Test Virtual Machines (TVM)* that defines how the test should start, end, or map memory. The second suite defines 173 tests to check the compliance of a RISC-V device to the specifications. It checks the main aspects of the implementation to make sure the specification has been interpreted correctly. The modified version of CVA6 passes all tests from both suites.

5.4.2 Experimental Setup

To measure the impact on the performance of the executed code, we use Gigue to generate the same workloads, either using the base execution model or the JITDomain framework. The values used for the application classes are derived from the benchmarks' execution analysis presented in the previous chapter, Section 4.4. We implement the JITDomain software solution in Gigue.

Gigue Implementation

To determine the performance overhead of the solution, we add it to Gigue as a new `Builder` along with the custom instructions. We duplicate all loads and stores (`lb1`, `lh1`, `lw1`, ..., `sb1`, `sh1`, `sw1`, `sd1`), and add the two domain-changing instructions for calls and returns changing domains (`chdom` and `retdom`). To separate the interpretation loop in one domain and the JIT code in another, all generated loads and stores in the JIT code are set to their duplicated versions. The prologue and epilogue of all methods in the JIT code use `sst`/`lst` to store and load return addresses to the shadow stack. All calls from the interpreter are modified to use `chdom` as their control-flow transfer instruction. They

2. `https://github.com/riscv-software-src/riscv-tests`
3. `https://github.com/riscv-non-isa/riscv-arch-test`

target the `call` trampoline that pushes the incoming address on the shadow stack, sets the return address to the `return` trampoline and redirects the control-flow to the initially requested JIT method. On returns to the interpreter, the `return` trampoline pops the return address from the shadow stack and defers the control-flow back to the interpreter using `retdom`. Overall, the code size overhead boils down to the addition of a decrement and increment of the shadow stack pointer in the prologue and epilogue of every JIT method.

Core Setup

We ran the workloads under Ubuntu 20.04.6 LTS, on an Intel Core-i5 machine equipped with a 1.60GHz CPU and 16GB of memory. We use the CVA6 commit `bb80b3f` (in the absence of proper version tags), Verilator version 5.008, and execute binaries linked and disassembled with the RISC-V GNU toolchain version 2.40.0. We use the same application classes as the ones we defined in the last chapter: varying call occupations between 1% and 6% of instructions in method bodies, and varying memory access intensities between 4% and 20% of the instructions in method bodies. All generated binaries use the same seed as their corresponding un-instrumented counterpart, each configuration is generated and executed ten times on their respective core.

5.4.3 Experimental Results

Performance Results

We present both runs and comparisons between the baseline CVA6 core and JITDomain implementations in Table 5.3 and Figure 5.10 for the different "call" applications, and Table 5.4 and Figure 5.11 for the different "memory" applications. Both figures present the overhead in terms of the number of cycles (darker color, low hatch number) and CPI (lighter color, high hatch number).

The impact on performance in terms of the raw number of cycles taken to execute the full binary is negligible as it averages at 1.51% for calls, and 1.27% for memory accesses. It is the most noticeable (at 2.4% for calls, 2.2% for memory accesses) when JIT methods are small, highlighting the additional decrement/increment of the shadow stack pointer in the prologue/epilogue of JIT methods. The average number of cycles per instruction is also affected to a minimal extent, averaging 0.95% for calls and 0.61% for memory accesses. Among all scenarios, the performance impact is minimal.

Table 5.3 – Performance overhead (raw number of cycles / cycles per instruction).

Call occupation \ *Method Size*	1. 400 bytes	2. 600 bytes	3. 800 bytes
1. 1% of instructions	2.43% / 2.34%	1.67% / 1.47%	1.46% / 1.15%
2. 3% of instructions	2.29% / 1.52%	1.56% / 0.72%	0.98% / 0.29%
3. 6% of instructions	1.22% / 0.27%	0.94% / 0.22%	1.01% / 0.49%
Mean	**1.51% / 0.95%**		

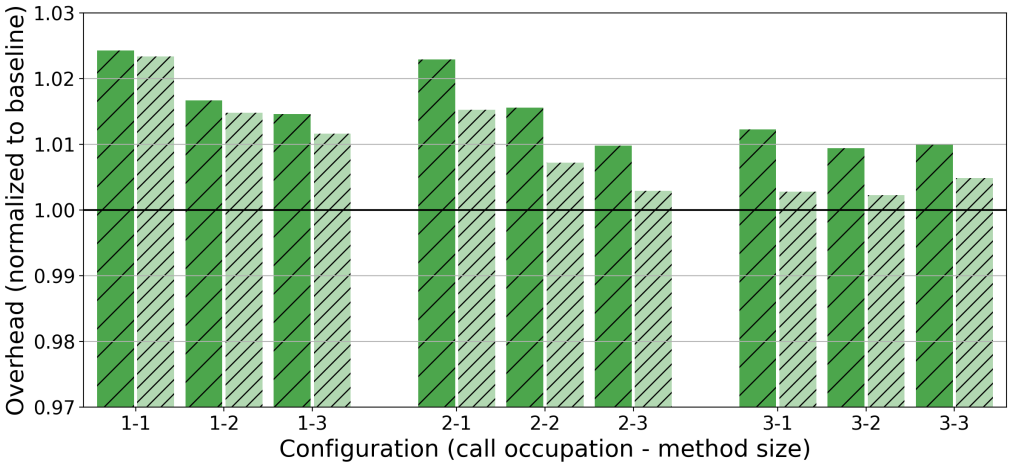
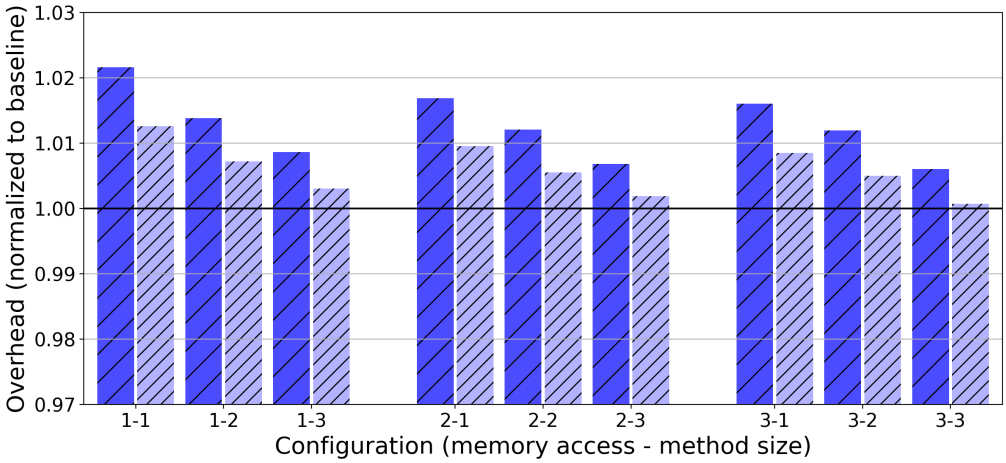


Table 5.4 – Performance overhead (raw number of cycles / cycles per instruction).

Memory Access \ *Method Size*	1. 400 bytes	2. 600 bytes	3. 800 bytes
1. 4% of instructions	2.16% / 1.26%	1.39% / 0.72%	0.86% / 0.31%
2. 12% of instructions	1.69% / 0.96%	1.21% / 0.55%	0.68% / 0.19%
3. 20% of instructions	1.60% / 0.85%	1.20% / 0.50%	0.60% / 0.06%
Mean	**1.27% / 0.61%**		



	LUTs		**Slice Registers**	
	Baseline	*JITDomain*	*Baseline*	*JITDomain*
Frontend	3190	3171	4058	4060
Decode Stage	609	664	266	269
Issue Stage	14936	15134	8983	9036
Execute Stage	23391	23410	7563	7571
Commit Stage	203	218	0	0
CSR Regfile	2434	2408	1691	1716
Cache Subsystem	6242	6242	2507	2509
Total	**52719**	**52961**	**25482**	**25575**
		(+0.46%)		**(+0.36%)**

Table 5.5 – JITDomain Resource Utilization

FPGA Resource Utilization

Regarding the FPGA resource utilization, Table 5.5 presents the impact of the JITDomain solution on the CVA6 core compared to its baseline unmodified counterpart. JITDomain incurs less than 0.5% of area overhead in terms of Look-Up Tables (LUTs) and slice registers. We estimate this area overhead to derive a negligible power overhead. In addition, none of the core modifications interfere with a critical path, leaving the maximum frequency untouched. Overall, the JITDomain implementation has a negligible area and power overhead and keeps the maximum operating frequency of the core.

5.5 Summary

In this chapter, we have presented JITDomain, a security framework that uses custom instructions to enforce domain isolation. We have presented its usage from the software use case to its hardware implementation in a fully-featured open-source core, CVA6. We implemented the solution by extending the PMP module with domain information and storing the current domain in a dedicated CSR. All instructions are tagged with domain information that is enforced at the decode stage for code execution, and at the memory access level for both instructions through the fetch stage and data in the load/store unit. These two checks and the tagging of instructions guarantee three main primitives: *call stack separation*, *JIT data isolation*, and *system call filtering*. We evaluated the implementation using Gigue-generated binaries that implement the custom instructions and the security model defined in previous sections. The impact on performance is minimal and corresponds to less than 2% additional cycles on average, and less than 1% CPI over all scenarios. The corresponding impact on FPGA resources is minimal as well with 0.5% area overhead. To encourage reproducibility of results, as well as usage and improvement of the solution, the JITDomain implementation code base is open-source and available on Github [4]. In addition, the assembly test suite along with its patched toolchain and runner is also available [5].

4. `https://github.com/QDucasse/cva6`
5. `https://github.com/QDucasse/jitdomain-tests`

Chapter 6

JIT Compiler Extension to RISC-V

By the start of 2021, no JIT compilers were ported to RISC-V in major VMs. To be able to experiment with custom instructions on this new ISA, we wanted to use a fully-featured VM. Major JavaScript virtual machines such as SpiderMonkey (embedded in Mozilla Firefox) or V8 (embedded in Google Chrome) were not available on RISC-V. The Pharo VM is an actively maintained virtual machine used in production and recently ported to newer architectures, such as ARMv8 [42], using an iterative test framework. We chose to extend its JIT compiler, Cogit, to the RISC-V ISA for research, to prototype the usage of custom JIT instructions in the future.

6.1 The Pharo Virtual Machine

The Pharo language is an evolution of Smalltalk according to the Smalltalk-80 specification. It is a pure object-oriented dynamically-typed programming language that revolves around message passing as its way to transfer control flow. It defends a simple syntax and extensions over the base Smalltalk language. Along with the language, Pharo is also defined by its live development environment that provides powerful tools to navigate, execute, and debug. As the environment itself is written in Pharo, any base object or tool is extensible and inspectable. The base environment containing the main tools is available open-source [1] as an "*image*". It has been in development for more than ten years and is used in production on a wide range of applications: from web development, data analysis, and visualization, to user interfaces. At their core, these applications are executed on top of the open-source Pharo Virtual Machine [2].

1. `https://github.com/pharo-project/pharo`
2. `https://github.com/pharo-project/pharo-vm`

6.1.1 VM Compilation

Meta-circular VMs: The Pharo VM is a meta-circular VM, in the sense that its source (the language it is *written in*) and target language (the language the runtime *supports*) are the same. It is a VM for Pharo written in a restricted version of Pharo, following the original design of the Squeak VM. Several other projects use dynamics and very high-level languages for interpreters and VM implementations and each one of them answers the bootstrap step differently. The idea comes from Common Lisp and Self [63] and, in the same vein as the chicken and the egg, writing the run-time support for a language in the language itself raises the question of the first run of a program. Jikes RVM (formerly Jalapeño [176], [177]) is a Java VM with a JIT compiler that bootstraps by self-applying the compiler on a host VM, then dumping a snapshot from the memory of the resulting machine code. Another JVM, the MaxineVM [178], [179] follows the same approach using a host VM to perform its initialization step. In their example, they both used the HotSpot JVM [59]. Other VMs use a restricted version of the language that is more easily converted into another language for the first compilation. This is the case of the Squeak VM [180] (Smalltalk), which is then translated to C using Slang, or the Pypy VM [181] (Python) that uses a restricted version of Python, RPython [182], that is then translated to several available backends. The main goal for meta-circular VMs is to take advantage of the high-level language constructs and tools during the development process.

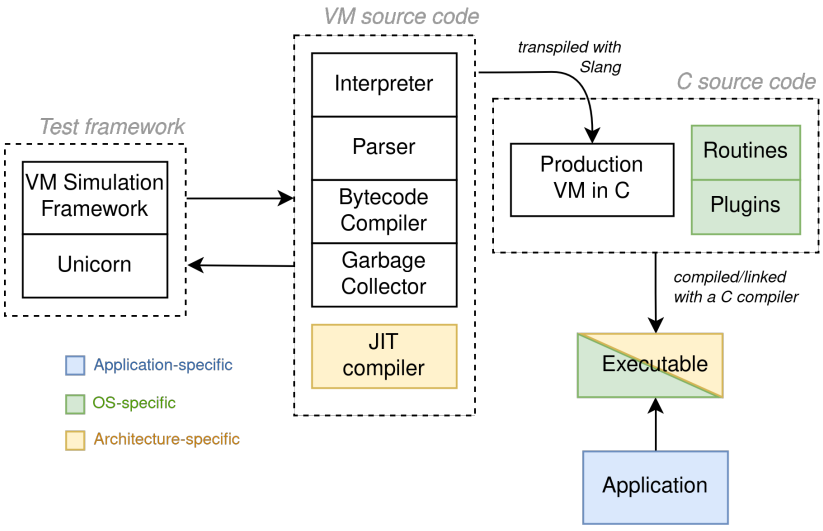


Figure 6.1 – Pharo VM Compilation Process.

Slang: The Pharo VM historically inherits from the Squeak VM [180] and is written in a restricted Smalltalk version and then transpiled to C using a VM-specific translator called Slang [180]. Slang translates a group of classes into a single C file, transforming methods into functions. Slang restricts the source language from some features such as polymorphism or exceptions to correctly translate it to C code. The C code is then compiled along with mandatory and optional plugins to an executable that runs on the desired architecture. During its translation process, Slang optimizes the interpreter-generated code by inlining the different bytecode cases and organizes the cases as threaded code following Ertl et al. guidelines [65]. This way, the main VM components, the interpreter, parser, bytecode compiler, interpreter, and JIT compiler are all written in Slang. The whole process is presented in Figure 6.1 and highlights the portability of the different parts, with the JIT compiler standing out as it has to be rewritten for new architectures.

6.1.2 VM Runtime

Applications and Images: As described by Miranda et al. [38], Smalltalk is an object system, where the entire system along with its development tools and application code is stored in a snapshot file called an *image*. This image essentially is a memory dump of the entire heap containing both objects, compiled methods, and running processes. At run time, the VM structures the memory in different spaces: `.text` and `.data` zone, the machine code zone (where the JIT code is installed), several segments for live objects, and both the native C call stack (for interpretation) and Smalltalk call stack (for JIT code). Developing applications consists of writing and editing code, which in turn installs, modifies, and removes classes and compiled methods from the class hierarchy. As everything is an object, the development environment itself can be inspected and altered on the fly. The application code can be extracted and distributed with a package manager, and interfaced with a version control utility. The memory layout is presented in the next subsection in Figure 6.3, compared with its simulation environment counterpart.

Compilers: The component responsible for source code compilation is *Opal*. It parses Pharo source code and translates it into an annotated Abstract Syntax Tree (AST) representation. The AST itself is then translated into an Intermediate Representation (IR) used for optimization purposes and designed to handle multiple sets of bytecode. Finally, the IR is translated into the chosen bytecode set. Once a method is detected as "*hot*" where it is frequently executed, the JIT compiler fires up and recompiles it to machine code. The Pharo VM implements a JIT compiler named Cogit [41] to speed up the ex-

ecution, which process, along with Opal, is presented in Figure 6.2. It does not model a control-flow graph and compiles at the granularity of a method linearly, meaning the generated machine code mostly has a one-to-one mapping to the JIT IR. No register allocator is defined as the CogRTL IR uses fixed virtual registers assigned ahead of time to physical registers for each backend. CogRTL is the intermediate representation Cogit uses to recompile the bytecode sequence it uses a 2-address-code intermediate representation (IR) called CogRTL to compile the succession of bytecode down to machine code. Cogit recompiles the bytecode down to machine code in three main steps: (1) a scanning phase looking for higher-level constructs (*e.g.* presence of calls implying the need for a call frame); (2) a bytecode parsing phase that translates the bytecode succession into the IR; and (3) a concretization phase where the IR is lowered down into machine code.

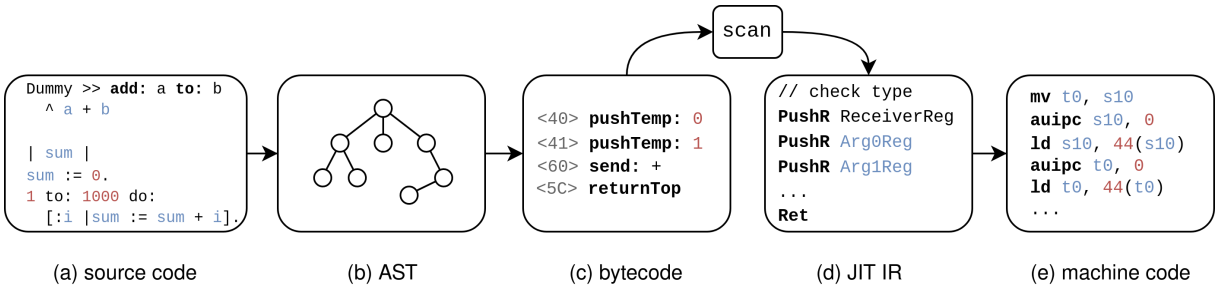


Figure 6.2 – Pharo source code compilation process.

Code patching: As Pharo is a dynamically typed object-oriented programming language, the JIT compiler also implements polymorphic inline caches [35] to improve performance, as presented in previous chapters. This means Cogit has to patch machine code without the knowledge obtained from bytecode scanning. Call sites are initially compiled as calls to dedicated trampolines send routines. The call site is then linked to the method itself once the method lookup is performed. Cogit uses machine code stubs to rewrite the needed hook points: a call to a trampoline becomes a type-checked entry point, a *monomorphic cache*. If the type check fails, a new case is added making it a *polymorphic cache* up until a threshold where it is upgraded to a *megamorphic cache*. This patching method is also used by the garbage collector to update references to moved objects. Since these patching methods use machine code stubs, they are architecture-dependent and need to be verified for each new architecture to support.

6.1.3 VM development process

As presented by Polito et al. [42] when presenting the port of the Pharo VM to ARM-v8, testing and debugging a VM is a hard task, as it has to deal with a wide range of functionalities and support both high-level and low-level mechanisms through the different components involved. The portability it provides has to be guaranteed on different architectures and operating systems. Taking advantage of the meta-circularity of the VM and being able to live code and debug the VM while it is being developed eases the otherwise tedious complete development cycle. As an example, the Maxine VM team reported their experience when extending their VM to ARM-v7 [183] on a QEMU-based architecture and doing heavy use of `gdb`, remaining low-level in the development state.

VM Simulation Framework: As the core components of the VM are written in Slang, the whole VM execution is simulated [38] by: (1) interpreting the Slang components directly in the live environment; (2) executing the JIT code generated using an external processor simulator; (3) simulating the memory using a large byte array to store the object spaces. This framework has several key advantages for VM development as it allows for deterministic simulation since the memory is not subjected to ASLR and uses its own clock to make VM bugs reproducible and debuggable. It allows for modularity in the simulated components for easier prototyping before choosing which version to ship and compile in the production VM. The simulated run-time memory is presented in Figure 6.3.

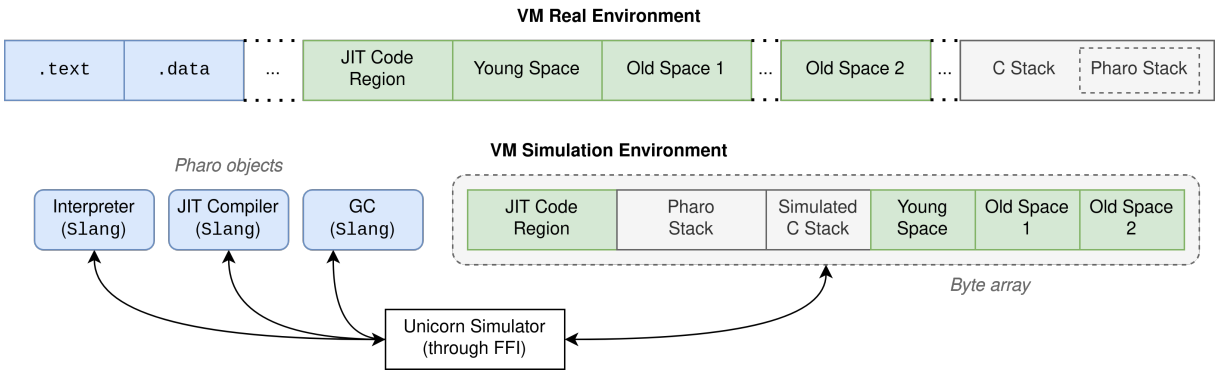


Figure 6.3 – Pharo VM simulation environment (from [38] and adapted to Pharo).

Test Harness: In addition to the simulation framework, a battery of unit tests have been added to the Pharo VM, around its JIT compiler to test the correct implementation of the compilation pass [42]. It works as a black box around the processor simulator,

where the bytecode set to recompile is passed to Cogit, the machine code is executed, and the side effects are verified. This harness was set up to speed up the port of Cogit to the ARMv8 architecture supported by newer Apple processors. It has been developed starting on a small scale, defining unit tests on specific tasks the JIT compiler should support, such as *bytecode compilation*, *correct header generation*, or *inline caches message sends* among others. It uses Unicorn [40] as its processor simulator, monitors the register state and catches any raised exceptions. Unicorn is a wrapper on top QEMU [43] that provides a variety of hooks to activate callbacks from the user side. The simulation version of Cogit maintains a set of dictionaries holding addresses to closures, or runtime routines, both sending messages to simulation objects. Cogit creates unique illegal addresses mapped to each of these variables or routines, using them as a key to the dictionary. When Unicorn catches an exception on these addresses, the corresponding simulation environment variable or closure is executed before sending the control flow back to the simulator.

6.2 Cogit RISC-V Backend Port

In this section, we present the port of the Cogit to the RISC-V ISA. The use of the previously introduced IR CogRTL makes the first two phases of the JIT compilation process agnostic from the ISA. However, we present patches that had to be introduced to perform correct mappings between the IR and machine code. Among them, we had to introduce new IR instructions better suited for the RISC-V branching mechanism. We extended the JIT compiler with RISC-V-specific IR instructions and defined new translations from these IR instructions into their corresponding machine code representation. We fixed the registers used by the Pharo VM according to the RISC-V calling convention.

6.2.1 RISC-V Design Choices

The RISC-V ISA enforces several design decisions to simplify the data flow and processor design. Patterson and Waterman present and motivate those decisions in the RISC-V Reader book [184], and among them:

Hardware and program size: Optional and composable instruction extensions allow a conservative support of instructions, that fit the program application and objectives. It directly impacts the size of the core with, for example, optional floating-point instructions (`F`) or multiplication and division instructions (`M`). A set of compressed instructions

(C) is also available to reduce code size for frequently used instructions.

Stack and branch instructions: There are no dedicated stack instructions on RISC-V, any push/pop to the stack has to be performed explicitly through a decrement/increment of the stack pointer and a corresponding store/load. Similarly, there are no dedicated call/return instructions that have to be handled using explicit loads and jumps from the link register with jal/jalr (*"jump-and-link"* and *"jump-and-link-register"*), along with the arguments set up in the 8 dedicated argument registers. To support unconditional jumps, those instructions can be passed the hardwired zero (x0) as the destination register instead of the link register (ra). The return instruction is also an alias of jalr with ra as its source register. These choices simplify the design of the processor centered around one dedicated multipurpose instruction for calls and jumps. For branches, RISC-V does not define condition codes and instead provides instructions to compare two registers and jump to an immediate offset directly. It also does not implement delayed loads and branches like the MIPS ISA or older RISC architectures where instructions after a branch or load instruction would be executed before the previous instruction has taken effect. The absence of those condition codes and delayed slots removes a complex hidden state from the processor pipeline and separates the architecture from the implementation.

	31 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	opcode
I	imm[11:0]		rs1	funct3	rd	opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
B	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
J	imm[20|10:1|11|19:12]				rd	opcode

Figure 6.4 – RISC-V Instruction Types Encoding.

Instruction homogeneity: The source(s) and destination registers are fixed in the instruction format, as presented in Figure 6.4. The encoding of the immediate values uses a constant size (20 bits for U and J, or 12 bits for the other types) and they are always sign-extended, effectively defining a single way to handle all immediate values from the processor point of view. Instructions are aligned to 4 bytes or 2 bytes for their compressed version and the program counter (pc) is *not* a general-purpose register. The stack pointer has to be aligned at the granularity of the address space, but memory accesses have their variants from byte to double-word accesses. All memory accesses load to/store from a

register through a single addressing mode: base register alongside a 12-bit immediate offset. The alignment requirement frees one bit off the encoding of branches and jumps offset as the last address bit is not considered.

Overall, these design choices simplify processor implementation but add complexity back to the compiler developer to correctly handle addressing modes, large immediate values, or operations found in other ISAs (*e.g.* arithmetic overflow check or implementation of rotation operations).

6.2.2 CogRTL and RISC-V

CogRTL Design: The CogRTL IR was designed with the x86/x64 architectures as the main targets with a nearly one-to-one mapping from the IR to the machine code instructions. This design choice followed the practice of *"optimizing for the common case"* which was the ubiquitous Intel x86/x64 ISA. As a result, JIT compiling to this architecture was simple, and it also worked for ARMv7 and ARMv8, which remained close to Intel, either for the number of registers, the many addressing modes, or the presence of flag registers and their associated branches and jumps. As a consequence and in light of the RISC-V shifts from traditional vendor ISAs, the CogRTL and RISC-V mapping become less evident.

Condition Codes:

The absence of flag registers and the lack of condition codes for branching has several implications. First, the IR to machine code mapping changes as two IR instructions result in a single branch instruction. To change the mappings, we add a notifier to the generation of branches and capture the following IR instruction during the IR lowering phase. We then generate a new IR branch instruction, defined only for the RISC-V ISA that allows for comparison between two registers. This keeps the bytecode scanning and IR generation phases identical for all backends and acts as a simple patch over architectures that do not implement condition codes.

Examples are presented in Figure 6.5 of the usual compare (`CMP`) instruction that, in architectures implementing condition codes, subtract the contents of the operand registers and discards the result, effectively setting up flag registers for a latter jump. We extract the operands of the first instruction, remove it (transforming it to a `Label` CogRTL instruction) and replace the conditional jump with its corresponding branch counterpart,

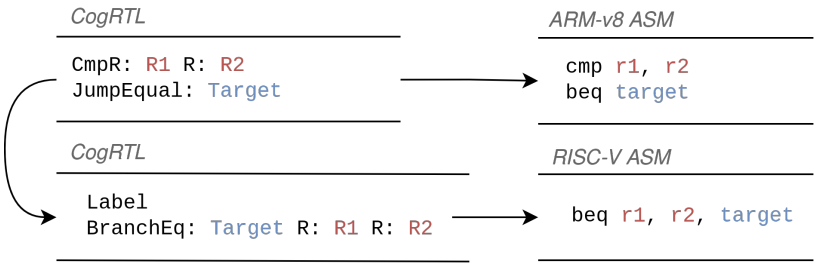
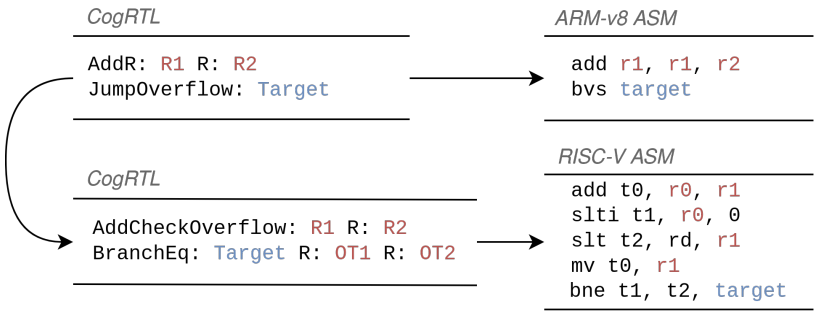


Figure 6.5 – Example of intermediate representation rewrite for conditional jumps, changing a 2 address code setting internal flag registers into an explicit comparison and branch for two registers.

fed with the extracted operands. CogRTL defines *"quick immediate values"* that the compiler tries to embed in the available space of `I`-type instructions before resulting in a load from an offset in memory. In this case, the constant is loaded into a temporary register, and the branch is operated between the temporary register and the one present in the comparison.

Figure 6.6 – Example of intermediate representation rewrite an arithmetic operation and an overflow check. A dedicated sequence of operations is emitted in machine code using temporary registers.



Additionally, as one of the expected condition codes is the check for arithmetic overflow, we now need to implement an overflow detection mechanism. We define new IR instructions for arithmetic operations with the added check for an overflow, performed directly in machine code, as presented in Figure 6.6. Those overflow checks (`JumpOverflow`/`JumpNoOverflow` CogRTL instructions) are used in the generation of arithmetic primitives [3] and in a special arithmetic bytecode.

Immediate values handling:

As presented earlier, the immediate value handling is centralized through RISC-V instructions. All immediate values are 12-bit (`I`, `B`) or 20-bit (`J`, `U`) long and are *always*

3. Routines compiled Ahead-of-Time (AOT) in machine code, for acceleration or OS interface.

sign-extended. Two common scenarios have to deal with wider immediate values: function calls and large immediate values loading.

Function Calls: In RISC-V, function calls operate around the link register `ra` and the use of the `jal` and `jalr` instructions. The first instruction takes a 20-bit offset, stores the next program counter in a register (`ra` by default), and jumps to the new `pc` and the sign-extended offset. It performs a pc-relative jump in a 1MB range. For any call beyond that range, `jalr` adds a 12-bit offset to a value in a register, stores the next program counter in a register (`ra` by default) and jumps to that value. It has to be used along with another instruction that handles the upper bits of the offset, `auipc` (*"add upper immediate to pc"*), to perform a complete pc-relative call. An important point is that both instructions sign-extend their respective immediate value and additional care has to be provided to handle the cases where the truncation of the offset in a low and high part activates the sign bit. This also has to be checked for pc-relative loads, the succession of `auipc` and `l<b|w|...>`. The issue has to be handled on top of the already existing translation between the representation of integer Pharo uses and what the underlying architecture expects. The process is presented in Figure 6.7 for both calls and PC-relative loads. The sign bit of the lower part of the offset is canceled by adding `0x800` to the full offset before extracting its upper 20 bits, effectively reversing the sign extension of the 12-bit lower immediate value.

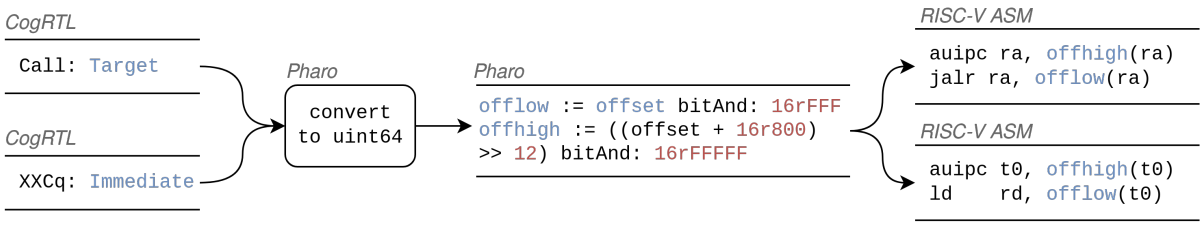


Figure 6.7 – Immediate Value Handling.

Large Immediate Loads: To load an immediate value in a register, RISC-V assembly uses the pseudo-instruction `li` (*"load immediate"*), that extends into 1 to 8 instructions (for 64-bit immediate values). This is handled through several concurrent policies that are scored in `gcc` or through a complex recursive function in `llvm`. The recompiled methods might require to be patched as presented before, and keeping space for the worst case brings a large code size overhead. Cogit uses *"quick immediate values"* to try and embed values in the available instruction space. A dedicated space is allocated for each recompiled method to hold large immediate values that are accessed through pc-relative

loads, resembling the `.text` and `.data` ELF segments. A manager handles the generation of this zone and guarantees any large immediate value takes at most two instructions to load.

Addressing Modes

The RISC-V ISA provides a single addressing mode to access memory, which is the use of a base address register and a 12-bit sign-extended offset. In contrast, x64 and even ARM-v8 provide additional ways to access memory, using a base register, a *displacement* (offset) and a *scale* value, power of two, multiplied to an index register. As an example, memory accesses in the form of `[regb + regi * scale + displacement]` are allowed using the `lea` (Load Effective Address) instruction on x64. Similar instructions are found in the CogRTL set and are supported translated in RISC-V machine code using a temporary register to define the final address of the access, resulting in at least two additional machine code instructions.

6.3 Tooling Extension

This section presents the set of tools, either internal or external to the VM development environment, that were used to port the JIT compiler. These tools have permitted the development of the RISC-V backend without access to the target architecture through a vendor processor.

6.3.1 Choice of the simulator

We initially started the development of the JIT compiler using the Spike simulator from which we extracted a shared library with the main functions we needed to map program memory, execute it, and get the register state. We managed to use it for the first tests on simple bytecode compilation. However, the lack of unified exception handling and hooks similar to QEMU/Unicorn made it complex to handle more complex tests that require several exception catch-and-resume as required by the simulation environment. The Spike extension and Pharo bindings are available open-source [4] and we kept this option for custom instructions as their inclusion in Spike is simple and standardized.

4. `https://github.com/QDucasse/spike-lib`

With the release of Unicorn2 beta, we switched to this processor simulator, updating the simulator bindings to integrate the RISC-V ISA, and performing the switch for other backends. Through development, several bugs in the simulator were pointed out, presented through unit tests, and corrected upstream. Among them are the activation of different RISC-V extensions, a bug in the PC update, or precision in self-modifying code (which we need for JIT code patching). Overall, 28 unit tests were added to the Unicorn project, through different backends (x86/x64, ARM32/ARM64, and RISC-V32/RISC-V64).

6.3.2 Test harness extension

To handle and verify the different issues highlighted by the translation from CogRTL to RISC-V machine code, we extended the test harness with RISC-V-specific tests. Those tests cover different aspects of the JIT compilation process. First, simple tests relative to the generation of correct RISC-V instructions and their encoding (1). These were written during development to guarantee all used instructions (around 80) have the correct encoding by simply checking the output of the `lldb` disassembler. Next, we test the correct result of mappings from IR to machine code (2), especially in cases where RISC-V and CogRTL do not match directly. We add tests to check the correct translation of conditional jumps to branches, as presented earlier. We also check the handling of different sizes of immediate values in both *calls*, *jumps* and *loads* for correctness through the sign-extension checks. Additional tests for the translations where no instruction is directly available for RISC-V are also added. Combinations of machine code instructions that handle arithmetic overflow or rotations, as well as the varying addressing modes, are addressed separately and executed to verify their soundness (3). Overall, around 100 tests are added to validate RISC-V-specific needs.

6.3.3 Machine code debugger

The processor simulator is used in black-box by the test harness that verifies the correct register and/or memory state once the tests are completed. While we can still inspect the large byte array Unicorn uses as memory, the increased code size and the IR-to-machine-code mapping no longer respecting a one-to-one ratio, it has become harder to inspect and investigate for bugs in the generated machine code. The Pharo VM development environment integrates a JIT code debugger that presents the state of the stack, registers,

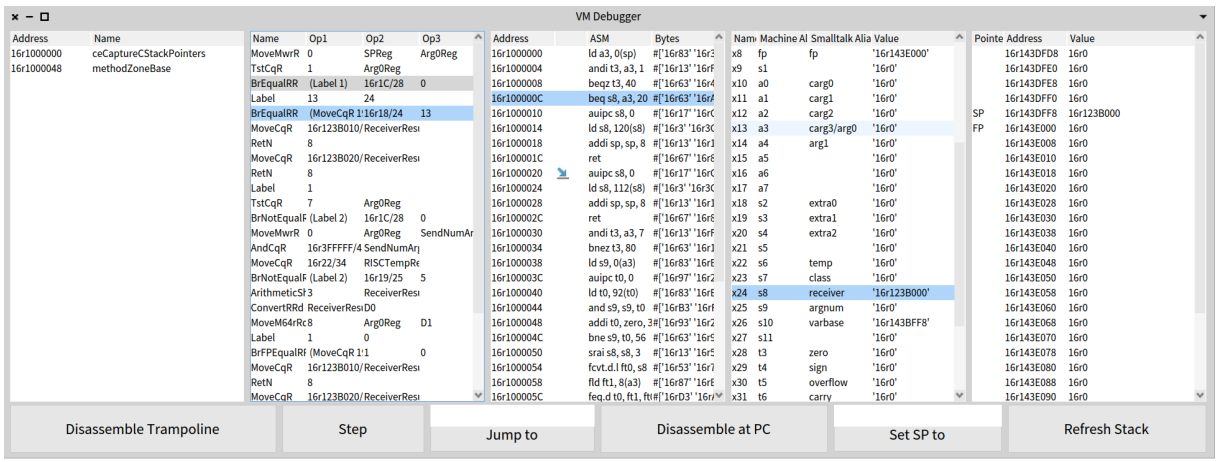


Figure 6.8 – Pharo machine code debugger.

and disassembled machine code. It allows for step-by-step execution as well as stack and register state inspection.

To better navigate RISC-V machine code, we extend its functionalities and add the support for trampoline disassembly and the corresponding CogRTL IR. To get program code information for RISC-V, we also extend the `lldb` Pharo bindings to support the RISC-V architecture. Finally, the correspondence between CogRTL, machine code, and affected registers is shown by highlighting the linked elements from the different columns. The prototype of the new layout that integrates the IR instructions is presented in Figure 6.8. When clicking on an IR instruction, the machine code debugger highlights the corresponding generated machine code instructions (third column) and the affected registers (fourth column). These additions helped us quickly navigate and point out bugs in failing tests.

6.3.4 Custom Instructions

To handle custom instructions in the VM development environment, we use a similar handler to the one presented in the Gigue testing framework. When the processor simulator handling JIT code reaches a custom instruction, it raises an exception triggering a handler hook. This hook extracts the instruction opcode and operands, checks if a corresponding simulation method exists and calls it.

Rotation simulation: As an example, we add a custom rotation operation, part of the (not yet ratified) `B` extension. As the rotation instructions are not available by default, we use 4 corresponding RISC-V instructions to perform the rotation. Moreover,

CogRTL defines the instructions `RotateLeftCq:R:` and `RotateRightCq:R:` as its means to shift the content of register by a shift amount passed as a *"quick value"*. We modify both translations as machine code into the emission of a single rotation with an immediate (`rori`/`roli`) when the immediate is embeddable in 12 bits, or load the value in a register and use the register rotation instructions otherwise (`ror`/`rol`). We define a hook on an "unknown instruction" exception, verify the opcode and perform the desired rotation in Pharo directly, before resuming the execution. This extension serves as an example for future custom instruction implementation and experimentation.

JITDomain simulation: Using the same custom instruction handling mechanism, we add partial support for the JITDomain framework by adding and verifying domains in the simulation memory. We attribute domains to the different parts of the memory layout: the machine code region is set to *"jit domain"*, the young and old spaces are attributed the *"base domain"*, alongside the C stack, and the Pharo stack is set to *"stack domain"*. We use a simulated *current domain* to check the control-flow follows the expected path, checking the *code domain* tag of executed instructions. We verify the *data domain* using the position of the memory access in the large byte array that is used to map memory in the simulation environment (see Figure 6.3).

All machine code is set in the *jit domain* where, as a reminder, data within it can only be accessed through duplicated loads/stores. We add the duplicated loads/stores to the generated JIT code that loads the PC-relative immediate values ("*quick values*"). This way, data embedded in the JIT method is only accessible through its method body. All CogRTL instructions that handle "*quick values*" use a pc-relative load (`auipc`/`ld`) that is changed to its duplicated counterpart (`auipc`/`ld1`). Loads and stores to outer memory are left untouched. We trigger Unicorn hooks from the machine code region on all duplicated and non-duplicated memory accesses to verify the correct domain of their accesses. For duplicated memory accesses, and after checking the domain, we perform the expected operation on memory and resume the execution.

For calls to the JIT code, as the interpreter itself is the real Pharo object, we change the current domain artificially when sending the control-flow to the machine code region. This mimics the action of a `chdom` instruction, effectively transferring the control-flow and changing the current domain to the *jit domain*. We also instrument the trampoline responsible for the callback to the interpreter (`returnToInterpreter`) to change domains through the `retdom` instruction, effectively switching between the *base domain* and *jit domain*.

6.4 Validation & Evaluation

We validate the port of Cogit to the RISC-V ISA by showing its compliance with the test harness, and by testing the speedup over the base interpreter version. The next steps for its total validation include the execution of all Pharo core tests on a vendor processor and would require a more robust cross-compilation toolchain and deployment process.

Functional Validation: As presented earlier, the Cogit RISC-V backend is integrated into the Pharo VM test harness. It validates all 800 present tests and the architecture-specific ones that were added. The port was developed starting from Pharo-X, and the tooling extensions were integrated into later versions of the Pharo VM. In addition, the port of Unicorn to its version 2 in the Pharo testing environment was propagated to all other backends as well, without incurring regressions.

Evaluation: We use a cross-compiled version of the VM with restricted features for a RISC-V architecture. We disable the optional support for Foreign Function Interfaces (FFI), network and socket support, graphics and fonts, as well as version control. We then build the VM executable along with the necessary shared libraries and deploy it on a QEMU Fedora image, version `20200108.n.0 Developer`. From there, we use a minimal Pharo image, stripped from optional external dependencies and plugins, and execute micro-benchmarks to extract the number of bytecode and the number of sends executed per second.

Micro-benchmark	Interpreter-only	With Cogit	Speedup
Bytecode/second	5 183 507	211 988 023	**40.89x**
Sends/second	232 685	14 752 081	**63.39x**

Table 6.1 – Cogit Microbenchmarks Execution and Speedup

We perform 10 runs of each micro-benchmark, each compared to a baseline VM using the interpreter only. In Table 6.1, we present the results of the experiments, with each column containing the geometric mean of the results of the ten runs. The version of the VM using the Cogit compiler executes 40 times more bytecode and 63 times more sends per second, showing a net speedup over the baseline interpreter.

We also investigated the impact of the backend on the size of generated methods and inline caches. Using the same setup and after running the previous micro-benchmarks, we disassemble the generated JIT code region to extract the number of elements as well as their mean size. The results are presented in Table 6.2 along with an x64 VM integrating

Architecture	JIT Code Size	Methods	PICs	Mean Method Size
x64	137 593 bytes	2992	747	**37 bytes**
RISC-V	142 389 bytes	2191	384	**55 bytes**

Table 6.2 – Cogit JIT code region comparison.

Cogit and executing the same tests. The results shown in the table present the average value for the 10 runs of each micro-benchmark. The impact of the mismatch between CogRTL IR and the RISC-V ISA is highlighted through the mean method size increase as it implies a code size overhead of 49%. RISC-V does not provide an extensive instruction set, to simplify processor design. On the other hand, Cogit uses an intermediate representation that was initially designed to target the dominant x86/x64 ISA and inherits its complex instructions, patched using additional RISC-V machine code instructions and JIT compiler passes, leading to an increase in the generated code size.

6.5 Summary

In this chapter, we have presented the port of the Cogit JIT compiler, embedded in the Pharo VM, to the RISC-V ISA. We have presented the main steps of compilation Cogit uses to translate Pharo bytecode into its corresponding machine code. We have shown how the design of the Cogit intermediate representation is tied to the x86/64 ISA and needs additional work to be compatible with RISC-V. The RISC-V ISA itself inherits from the RISC family and implements design decisions in its ISA that simplify processor design. This complexity has to be handled on the compiler side to implement similar capabilities to corresponding CISC instructions, at an important cost in code size overhead. To support and validate the addition of a new backed, we extended several of the tools already available in the Pharo VM development environment. Among them, the processor simulator - along with support for custom instructions -, the unit test harness with architecture-specific tests, and the machine code debugger to investigate more easily correspondences between IR and generated machine code. The RISC-V port validates all tests from the test harness and presents an average speedup of 40x in terms of bytecode executed per second, and 63x in terms of sends per second over its interpreter-only counterpart. The RISC-V backend is waiting for commodity hardware availability to perform the last tests before its upstream integration. The complete code of the port is

available open-source on Github [5] and is prepared to be integrated upstream. The port of Cogit to the RISC-V ISA opens up the door to further analysis on the concrete application of the JITDomain solution presented in the previous chapter, or the application of defense requirements SR1-SR5.

6.6 Next Steps

In this section, we present the next steps that will guide VM exploration for security along the requirements SR1-SR5. Regarding JIT code, Cogit uses a temporary memory buffer to generate the CogRTL IR from which it then generates the corresponding machine code. This buffer, along with the bytecode should be isolated in memory following SR4. Cogit then works at the granularity of a method and uses an explicit change from executable to writable to install the newly generated machine code. All accesses are explicitly performed using the permission change to: install a machine code method, extend/patch inline caches, compact JIT methods, link/unlink message sends in existing JIT methods, or mark elements for garbage collection. These elements validate part of SR1 requirements for writable/executable separation.

The data present in the JIT code region is handled by a `LiteralsManager` that stores them in a dedicated section after the body of the method itself. While this guarantees the proximity of the required data along with its dedicated method, it prevents clear isolation of JIT code and data, thus refuting SR2 (JIT data separation and restricted rights). A solution would be to dedicate part of the JIT code region to JIT data exclusively. This makes it possible to grant different permissions to both regions, effectively setting up JIT data to read-only, as it contains sensitive information such as jump target addresses. This is also a requirement for the complete adoption of JITDomain over the JIT code region.

Requirements SR3 (static code protection) and SR5 (sensitive object protection) have not been tackled by this thesis and would require additional work to design security validation and primitives around them. The port of the Cogit compiler to RISC-V opens up the door to further experiments with dedicated instructions, or implementation of existing security solutions.

5. `https://github.com/QDucasse/opensmalltalk-vm/tree/riscvX-conditional`

Chapter 7

Conclusion and Future Works

Language virtual machines come with a large set of interesting properties that have led to wide adoption. As so, they also constitute a perfect target for attackers. This thesis examines the various types of attacks to which a virtual machine in a language is exposed and proposes ways to protect and evaluate the effects of hardware-assisted solutions. This chapter provides a summary of the contributions of this thesis, examines the limitations of the tools and techniques used, and suggests potential areas for future research.

7.1 Summary of Contributions

This thesis examines the development of attacks on virtual machines (VMs), but more specifically highlights the importance of the Just-in-Time (JIT) code region in terms of both security and performance, as it is the place where fast machine code is installed and executed on the fly. Three types of attacks are described: code injection, code reuse, and data-only attacks. The first two either inject or build a shellcode from the executable part of the JIT code region. This is made possible by the predictability of both the JIT compiler output and the location of the JIT code. Data-only attacks are distinct, yet have the same objective: they concentrate on the inner workings or inputs of the JIT compiler to introduce a shellcode that eventually appears in the JIT code too.

Defenses evolve around VMs on three axes: diversification, memory protection, and capability containment. Using diversification and randomization of the JIT code, the predictability of its output is undermined. Dividing memory accesses for execution and writing is essential to guarantee `W^X` and prevent malicious code from being installed in the JIT code region. Capability containment comes in several forms to minimize the capacities of the untrusted executable code while keeping the impact on performance low. Control-flow integrity guarantees the correct execution of the program, while sandboxing isolates the untrusted code completely. In their most secure versions, both suffer from a prohibitive cost in performance. We motivate the use of hardware-assisted solutions that

provide strong security guarantees while incurring reasonable performance overhead. We also motivate the need for open and dedicated solutions, especially based on the RISC-V ISA since it allows for prototyping on fully-featured cores and reserves space for extensions in its standards. We defend the idea of custom instructions embedded in the JIT code that enforce isolation or control-flow integrity principles directly.

No solution can reasonably come without objective metrics, which, in return, can only be applied over test cases. The test cases come mostly from test benches that are static and hence offer no domain-space exploration facility. Instead, one can wish to benefit from non-real yet realistic test cases. Several metrics have to be taken into consideration: security guarantees, performance impact on the software run time, and resources used by the implementation of the design. Tooling must exhibit productivity gains, as an example, by working at a higher level of abstraction and offering extra services such as ease of inspection while preserving legacy compliance. They must also demonstrate that their behavior matches the specification.

To shrink the technological stack ranging from VM and JIT compiler development to processor support for custom extensions, we propose Gigue, a parametrizable random workload generator, completely compatible with simulated cores. It generates a workload whose execution model resembles the usual interpretation loop and calls to JIT methods. It is highly modular to give access to JIT code generation at different levels in the method body. It implements solutions from the literature along with the JITDomain method presented and allows for direct comparison between implementations. In addition to the generator, a test harness guarantees the sanity of the generated executables. By providing a pluggable API for custom instruction handling, Gigue also provides testability of custom instructions along with their application in real executable binaries. We also claim that defining the execution policy of custom instructions on the software side, through the processor emulator, helps outline the needs of the processor. An example use case, which defines different application classes with varying call occupations and memory accesses, is presented and executed unaltered on top of the Rocket and CVA6 processors. They provide a basis to compare instrumented versions using custom instructions whose correct execution is already guaranteed by the test framework.

The aforementioned tooling allows one to stress protection mechanisms. To guarantee the integrity of the JIT code, we define JITDomain, instruction-level domain isolation whose principle is taken from embedded device isolation models such as RIMI/DEMIX [33], [34]. We define duplicated memory access instructions and dedicated domain change

instructions. It extends the PMP defined in the RISC-V standards with additional domain checking and uses it to enforce three policies on the JIT code: filtering of system calls, JIT data isolation, and defines a JIT code shadow stack.

We demonstrate the efficiency of JITDomain by instrumenting Gigue binaries with the solution and adding support for the solution in the CVA6 processor. The changes in the processor require an additional layer on top of the PMPs (both code and data) and domain checking in the decoder. In addition to the implementation, we provide an assembly test suite to verify the correctness of the implementation. We evaluated the solution using multiple classes of applications generated by Gigue, presenting a limited impact on performance. The average overhead in terms of performance is averaged at less than 2% in the number of cycles and less than 1% in CPI over all scenarios. Although the implementation of the solution requires changes to the processor, they remain minimal, only leading to a 0.5% of area overhead.

Still, an in situ extensive validation requires one to apply the proposed solutions to a real use case, in our case, to a JIT compiler to a platform embedding our architectural proposals. The motivation we had to test the solution on a complex VM, led us to extend the Pharo VM JIT compiler to support the RISC-V ISA. As a result, the Pharo community was presented with a new execution platform, which has sparked a growing interest.

This extension requires an adaptation of the JIT compiler intermediate representation to correctly translate down to RISC-V instructions. Existing development tools were extended to support the new ISA. Among them, the in-image machine code debugger (offering high productivity) was extended to support more visualizations and the new ISA. The processor emulator was also extended and patched upstream for RISC-V correctness. Similarly to Gigue, custom instruction handlers have been added in the Pharo VM testing environment. Using a RISC-V Qemu image, the RISC-V JIT compiler was shown to be 40x/63x faster than the corresponding interpreter in terms of bytecode/calls per second.

7.2 Limitations

Although Gigue allows for quick and varied generation of application classes, the random nature of the workload it generates may not be fully representative of application execution. Although we believe that using the distribution of instruction types, once the JIT code has reached a stable state, should provide an interesting insight into the impact

of a custom instruction, it does not replace its final usage in a VM. In addition, Gigue is a workload generator and, while implementing custom instruction software support in its test framework, it does not generate a test suite for a given solution. It is meant to work along with a test suite to assess the impact and correctness of a solution. Its execution model and binary structure are based on the JIT code region of the Pharo VM, which uses a non-optimizing linear JIT compiler and confines the accesses to and from its JIT code region through dedicated trampolines. While this simplifies the adoption of the JITDomain model, it may not be fully representative of virtual machines that use several increasingly optimized JIT compilers.

The JITDomain execution model enforces a simple but effective isolation model. It guarantees several policies (system calls prohibition, data access control, shadow stack definition), but requires that the `chdom` and `retdom` instructions are used only by the VM when calling and returning from the JIT code. In the Pharo VM, trampolines are responsible for switching from the interpreter to the JIT code and are the only ones instrumented with these instructions. The trusted computing base (TCB) of the solution has this requirement and also expects careful setup of the `pmp`*`i`*`cfg`/`pmpaddr`*`i`* CSRs for PMP regions and the `dmpcfg`*`i`* CSRs for their related domain. As it is based on the PMP, the number of domains is limited by the number of PMP regions supported in the processor. We presented the solution with a restricted number of domains: Only two code domains and three data domains were defined for our needs, namely the base code/data domain, the JIT code/data domain, and the shadow stack data domain. Additional domains may be needed for other applications and more complex usages. Finally, hardware support for the solution requires an extension of a processor that performs invasive changes in the core, be they even minimal. With Gigue in mind, we want to evaluate the benefits of this type of approach compared to coprocessor solutions that are less invasive to the core, following already-defined extension APIs but might add additional overhead.

7.3 Future Works

Despite the few limitations the former section itemizes, this work led to many improvements to existing solutions, offers a parameterized generator that can be further reused in different contexts and by different kinds of user, validates protection schemes on a real platform, which is called to play a major role in the next year for embedded systems, and evaluates the pros and cons of these mechanisms by scoring their impact following a set

of representative metrics. Also, one important contribution of our work has been to port the Pharo VM, so that to support RISC-V.

However, this thesis opens several opportunities for future work in different domains.

7.3.1 Gigue and Hardware Development

In principle, Gigue generates workload, which calls custom instructions to test their impact at scale. However, we believe that Gigue could also be used more extensively to perform hardware test-driven development. The Gigue test framework provides an API to handle custom instructions and perform the expected actions on a software counterpart of the test. With the additional subprojects Prelude and Toccata, we define additional helpers to speed up hardware development. Prelude defines an API to guide the developer to (1) patch the `gnu` toolchain with custom instructions, (2) provide simple minimal examples (named *tutorials*) using the custom instructions. While Prelude is still in an early stage and does not replace the assembly test suite that is used to verify the implementation of a solution, we believe it should be extended to provide such tests along with complete binaries implementing the solution. These are presented as ① in Figure 7.1. On the other hand, Toccata is a workload organizer and runner. It works on top of Gigue, using varying parameters and interfacing with any RISC-V processor using a Verilator-simulated model. We believe that both subprojects could be extended to provide a hardware developer with a hands-on testing suite and benchmark runner to interface directly with its core.

We used Gigue to assess the impact of the JITDomain security solution, but another shadow stack solution in the test framework. The primary objective when Gigue was developed was to define a modular and parameterized generator, providing the capability for custom instructions to be integrated at various anchor points during the generation process. Few comparisons between hardware solutions are presented in the literature because they are often compared to their software counterparts. Gigue could serve as a common ground for comparison between different implementations of a solution, such as different shadow stack implementations [115] or domain isolation techniques, presented as ② in Figure 7.1. These comparisons are very valuable as guidance for future implementations and also as best-practice usage. Important metrics should cover both the software part through an increase in performance and code size, and also the hardware part through the area, power consumption, and invasiveness of the design. This comparison in the context of the modeled JIT code region generated in Gigue binaries would provide insight and

guidance on the choice of a hardware-supported security solution.

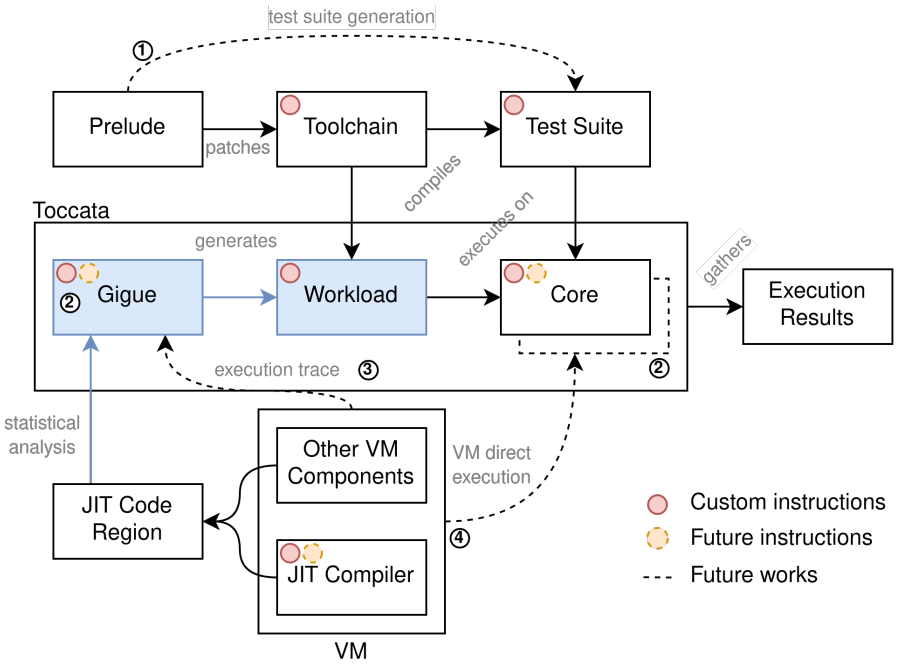


Figure 7.1 – Global view of the thesis along with projected future works.

Although the JITDomain security solution was tested on CVA6, an in-order core, it should also apply to out-of-order cores. However, support for the solution should require additional efforts. Tagging of memory access requests, whether from the instruction fetch (and I-Cache) or load-store unit (and D-Cache), is required for JITDomain to work correctly. The domain-changing instructions affect a CSR to store the current domain, propagated at the instruction commit stage. They flush the pipeline to isolate the JIT code completely during control-flow transfer. This mechanism complexifies out-of-order execution and requires careful attention when implementing it. A shadow stack solution defined using dedicated instructions, as presented in the JITDomain chapter, would be implemented in out-of-order cores without complex changes to the execution model. Extension to other cores is presented as ② in Figure 7.1. Looking in the other direction, Gigue can also be used to compare different solutions in smaller cores, especially since the need for VMs on embedded devices is increasing through their global adoption.

7.3.2 JIT-Specific Custom Instructions

As presented earlier, Gigue generates random workloads based on input parameters extracted from the stable state of the JIT code region. Gigue could also benefit from implementing other elements in the JIT code region it generates, for example, hidden classes, inline caches, and additional trampolines. We claim that these workloads give an interesting insight into the impact of custom instructions in the context of (simplified) JIT-compiled methods execution. However, using common execution traces extracted from a real VM would help to outline a "usual execution path" and propagate meaningful information in the Gigue binary. This idea is presented as ③ in Figure 7.1. We would like to extend Gigue by conserving its use of JIT methods, extracted from real VM usage, and compressing interpreter calls in the interpretation loop portion of the binary. Additional efforts are necessary to respect the Pharo VM calling convention and the internal usage of fixed registers. Its interface with Gigue should also be extended using the rich information that the Pharo JIT methods contain, *i.e.* metadata, call frame creation, method body, data, etc.

With the complete stack defining an extension, from the JIT compiler using custom instructions down to the processor, or coprocessor supporting it in hardware, the question of "what would be interesting custom instructions?" raises. We first need to complete the verification of the Pharo VM that implements the solution on a supported processor. This is represented as ④ in Figure 7.1. It requires additional modifications to the layout of the JIT code region in the Pharo VM, code and data for a given JIT method are placed next to each other, while JITDomain expects JIT code and JIT data to be fully separated. The authors of NoJITsu [9] present a way to split the memory layout between the JIT code, defined as executable only, and the JIT data, defined as read-only.

In a wider scope, the RISC-V `J` extension, currently in draft, is interested in "languages that are interpreted or JIT compiled, or which require large run-time libraries or language-level virtual machines" [185]. Its typical use covers garbage collection, dynamic typing, reflection, and dynamic dispatch. Any proposal to this `J` extension would benefit from having a synthetic workload that implements new instructions to assess their impact. Fortunately, security extensions could make their way in such an extension to directly protect VMs.

7.4 Final Remarks

In summary, we propose, implement, and evaluate a hardware-supported security framework to protect JIT code. It enforces security policies on the JIT code region, as expected by state-of-the-art defenses around VMs. In addition to the proposed solution itself, we propose an assembly test suite to ensure its correct implementation in the cores. To assess the impact of the solution and measure it against others, we also propose Gigue, a workload generator. We finally extended the Pharo VM JIT compiler to support the RISC-V ISA and support the first steps of the security framework. We believe that those three open-source elements provide a realistic environment for future evaluations and extensions in the fields of both hardware security and its application to language VMs.

Appendix A

Address (Re)Randomization

Motivation: Address Space Layout Randomization (ASLR) [186] is implemented at the OS level in OpenBSD and Linux, effectively randomizing the location of code and data in virtual memory. For each process, the base address of the executable, libraries as well as the stack and heap are hidden and randomized from an attacker. These addresses have to be guessed, at the risk of crashing the running application if they are mistaken. However, ASLR is defeated by JIT-spraying attacks, because the code region is filled with a `NOP` sled leading to the exploit. Enforcing back the OS-level idea on the generated JIT code would add complexity for an attacker. Randomization techniques either re-randomize the address space layout on a time basis or insert randomization components in the JIT compiler flow.

Re-randomization techniques [187]–[192] help defeat code-reuse attacks by adding a recurring constraint to an attacker. Ahmed et al. [193] compare solutions around fine-grained ASLR techniques under the constraints of JIT-ROP [5]. The authors conclude that fine-grained randomization techniques do not impose significant gadget corruption against a JIT-ROP threat model (see Section 2.2.3). They provide a way to determine the re-randomization upper bound for a given application (ranging from 1.5 to 3.5 seconds) to make randomization effective.

Runtime randomization is needed to randomize the JIT memory region. Librando [13] is a framework that applies to a JIT compiler in a black-box manner. The operating system memory allocation functions are intercepted to analyze, diversify, and rewrite the JIT code. All basic blocks generated are therefore randomized after the JIT compiler has generated the output machine code, but before its installation into memory. The contents of the calling stack are preserved, making the VM use it transparently without reworking the compiler internals. The performance penalty is high, as it comes with a slowdown of 1.15x on the JVM and 3.5x for V8. Isomeron [194] completes the defense by providing both code randomization and execution-path randomization under the JIT-ROP constraint. It does not require source code or a static analysis phase and works at

the basic block level. The authors measure the impact of execution path randomization on the static SPEC CPU2006 benchmark and measure 19% of average overhead.

JITGuard [10] authors store the JIT code in a randomized memory region whose location secret is stored in a hardware enclave alongside the JIT compiler itself (see details in Section 2.3.4). All memory accesses are zeroed out at initialization to leave no indication of the location of the JIT code region. Any function that needs to be JIT compiled goes through the enclave and the JIT compiler generates and installs it in randomized memory. At run time, indirections help jump to and return from the randomized region. Each JIT function has its dedicated trampoline that uses a segment register and a segment jump table to jump to the randomized position in memory. Both segment memory regions cannot be accessed by an attacker as they are hidden behind the `arch_prctl` system call that is not supported in Linux. More details on this solution are presented in Section 2.3.4.

Discussion: Code and data randomization help counter code-reuse attacks by making it harder to disclose the location of a code or data pointer in the memory. The OS-level ASLR solution is not sufficient to protect dynamically generated code and the VM in general. Even fine-grained randomization [195] does not suffice when faced with a JIT-ROP threat [5]. Re-randomization techniques allow a run-time application of memory space randomization at a heavy cost of performance (which can still be dimensioned following rules [193]).

Appendix B

Sandboxing

A primer on sandboxing: Sandboxes are execution environments that impose restrictions on system resources from applications. They come as a system call interposition at different levels (kernel, user, or both). Pure user-level sandboxes can be realized through software-based isolation techniques such as *e.g. Software Fault Isolation (SFI).* Strictly OS-based mechanisms reside entirely in the kernel (*e.g. hypervisor*) and rely on hardware memory. Delegation-based sandbox architectures (*e.g.* Ostia [196]) come as hybrid solutions and require the sandboxed process to drop most of its privileges and delegate sensitive operations to the trusted process. As presented in the next two solutions, SFI combines static analysis with software guards and enforces them on JIT code at generation, modification, installation, and deletion time.

Application to VMs: Ansel et al. [125] use now-deprecated Google Native Client (*NaCl*) [197], a user-level sandbox architecture based on SFI. In their setup (③ in Figure B.1), the JIT compiler is defined as *untrusted* NaCl code, generates machine code in a buffer, then invokes the interface with the NaCl trusted runtime to verify the generated machine code and install it. Other interface functions modify and delete code stubs as requested by the JIT compiler. The JIT code is separated from the JIT data to be properly instrumented. All generated machine code is validated by the NaCl runtime using SFI: the target code address is verified to be in boundaries with the NaCl memory and aligned, the code is then copied to the private memory of the trusted runtime, and it is verified using the NaCl validator, the target address range is checked to be unused, then reserved, and the code is copied to. While the solution proposes interesting security properties, it shows a high run-time overhead from 28% up to 60%.

Song et al. present *Secure Dynamic Code Generation (SDCG)* [17], a multiprocess-based architecture that uses a delegate-based sandboxing architecture (namely *seccomp* [114]). In their setup (② in Figure B.1), the authors take a different approach by defining the JIT compiler as the trusted process. The code cache is stored in shared memory that is mapped as `RX` in the original process and `WR` in the trusted dynamic code generation

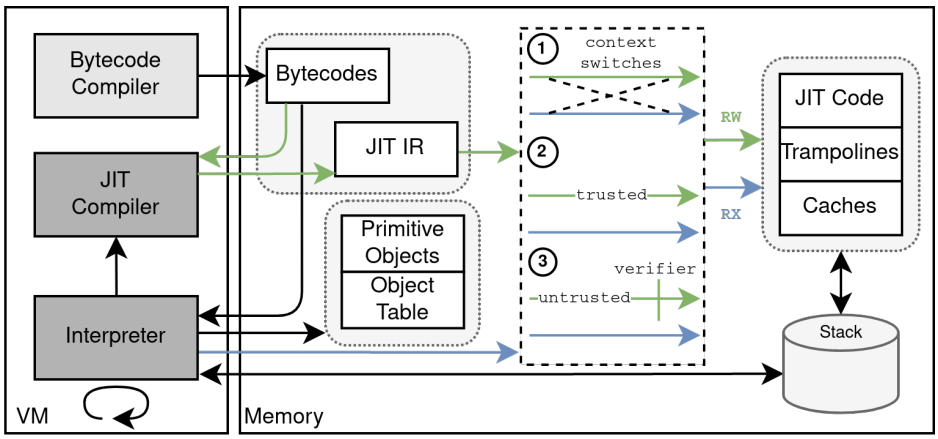


Figure B.1 – Sandboxing overview, ① represents a simple process isolation through context switches such as Lobotomy [16].

process. The code cache remains `R`-only in the untrusted process, while the JIT compiler can safely perform code generation, installation, patching, and deletion. The authors extend the idea of Lobotomy with a separate process that performs more than just JIT compilation. To keep the implementation transparent, wrappers are added to make the trusted process invocable through remote procedure calls. SDCG is built on a delegation-based sandbox architecture and uses it to catch system calls related to virtual memory management. SDCG enforces three important policies on caught calls: (1) memory cannot be mapped as both writable and executable; (2) when mapping a region as executable, the base address and size must come from the trusted process, and the memory is always mapped as `RX`; (3) the permission of non-writable memory cannot be changed. The solution applied to V8 adds about 11% of overhead while isolating the dynamic code generation process.

Discussion: Sandboxing helps cleanly isolate the JIT compilation process and/or dynamically-generated code, but comes with an important instrumentation cost and can induce a high overhead of more than 50% in the worst cases. It does not suffice against ROP attacks with and without a scripting environment and fails against data-only attacks where an attacker does not need to break out of the current process.

BIBLIOGRAPHY

[1] Codenomicon. (2014). The Heartbleed bug, [Online]. Available: `https://heartbleed.com/`.

[2] T. O. project. (1998). OpenSSL, cryptography and SSL/TLS toolkit, [Online]. Available: `https://www.openssl.org/`.

[3] F. Zaruba and L. Benini, « The cost of application-class processing: energy and performance analysis of a Linux-ready 1.7-GHz 64-Bit RISC-V core in 22-nm FDSOI technology », *IEEE Transactions on Very Large Scale Integration (VLSI) Systems (TVLSI)*, vol. 27, *11*, pp. 2629–2640, 2019. DOI: `10.1109/tvlsi.2019.2926114`.

[4] A. Sintsov, « Writing JIT-spray shellcode for fun and profit », DSecRG: Digital Security Research Group, Tech. Rep., 2010.

[5] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, « Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization », *in Proceedings of the IEEE Symposium on Security and Privacy (S&P'13)*, IEEE, 2013, pp. 574–588. DOI: `10.1109/SP.2013.45`.

[6] M. Athanasakis, E. Athanasopoulos, M. Polychronakis, G. Portokalidis, and S. Ioannidis, « The devil is in the constants: bypassing defenses in browser JIT engines », *in Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS'15)*, NDSS, 2015. DOI: `10.14722/ndss.2015.23209`.

[7] W. Lian, H. Shacham, and S. Savage, « Too LeJIT to quit: extending JIT spraying to ARM », *in Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS'15)*, NDSS, 2015. DOI: `10.14722/ndss.2015.23288`.

[8] ——, « A call to ARMs: understanding the costs and benefits of JIT spraying mitigations », *in Proceedings of the 24th Network and Distributed System Security Symposium (NDSS'17)*, NDSS, 2017. DOI: `10.14722/ndss.2017.23108`.

[9] T. Park, K. Dhondt, D. Gens, Y. Na, S. Volckaert, and M. Franz, « NoJITsu: locking down JavaScript engines », *in Proceedings of the 27th Network and Distributed System Security Symposium (NDSS'20)*, NDSS, 2020. DOI: 10.14722/ndss.2020.24262.

[10] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi, « JITguard: hardening just-in-time compilers with SGX », *in Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*, ACM, 2017, pp. 2405–2419. DOI: 10.1145/3133956.3134037.

[11] T. Wei, T. Wang, L. Duan, and J. Luo, « Secure dynamic code generation against spraying », *in Proceedings of the 17th ACM SIGSAC conference on Computer and Communications Security (CCS'10)*, ACM, 2010, pp. 738–740. DOI: 10.1145/1866307.1866415.

[12] R. Wu, P. Chen, B. Mao, and L. Xie, « RIM: a method to defend from JIT spraying attack », *in Proceedings of the 7th International Conference on Availability, Reliability and Security (ARES'12)*, 2012, pp. 143–148. DOI: 10.1109/ares.2012.11.

[13] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, « Librando: transparent code randomization for just-in-time compilers », *in Proceedings of the 20th ACM SIGSAC Conference on Computer and Communications Security (CCS'13)*, ACM, 2013, pp. 993–1004. DOI: 10.1145/2508859.2516675.

[14] P. Chen, Y. Fang, B. Mao, and L. Xie, « JITDefender: a defense against JIT spraying attacks », *in Proceedings of the 26th International Information Security Conference (SEC'11)*, 2011, pp. 142–153. DOI: 10.1007/978-3-642-21424-0_12.

[15] P. Chen, R. Wu, and B. Mao, « JITSafe: a framework against just-in-time spraying attacks », *IET Information Security*, vol. 7, *4*, pp. 283–292, 2013. DOI: 10.1049/iet-ifs.2012.0142.

[16] M. Jauernig, M. Neugschwandtner, C. Platzer, and P. M. Comparetti, « Lobotomy: an architecture for JIT spraying mitigation », *in Proceedings of the 9th International Conference on Availability, Reliability and Security (ARES'13)*, IEEE, 2014, pp. 50–58. DOI: 10.1109/ares.2014.14.

[17] C. Song, C. Zhang, T. Wang, W. Lee, and D. Melski, « Exploiting and protecting dynamic code generation », *in Proceedings of the 22nd Network and Distributed*

System Security Symposium (NDSS'15), NDSS, 2015. DOI: 10.14722/ndss.2015.23233.

[18] B. Niu and G. Tan, « RockJIT: securing just-in-time compilation using modular control-flow integrity », *in Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, ACM, 2014, pp. 1317–1328. DOI: 10.1145/2660267.2660281.

[19] C. Zhang, M. Niknami, K. Z. Chen, C. Song, Z. Chen, and D. Song, « JITScope: protecting web users from control-flow hijacking attacks », *in Proceedings of the 34th IEEE Conference on Computer Communications (INFOCOM'15)*, IEEE, 2015, pp. 567–575. DOI: 10.1109/infocom.2015.7218424.

[20] A. Waterman, K. Asanovic, and S. Inc, *The RISC-V instruction set manual, Volume I: unprivileged ISA*, 2019. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/tag/Ratified-IMAFDQC.

[21] Intel, *Intel 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[22] A. De, A. Basu, S. Ghosh, and T. Jaeger, « FIXER: Flow integrity extensions for embedded RISC-V », *in Proceedings of the 26th Design, Automation & Test in Europe Conference & Exhibition (DATE'19)*, IEEE, 2019, pp. 348–353. DOI: 10.23919/date.2019.8714980.

[23] S. Park, D. Kang, J. Kang, and D. Kwon, « Bratter: an instruction set extension for forward control-flow integrity in RISC-V », *Sensors*, vol. 22, *4*, p. 1392, 2022. DOI: 10.3390/s22041392.

[24] ARM, *ARM-v8.5-A Memory Tagging Extension*, 2019. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[25] ——, *ARM-v8-M Reference Manual*, 2021. [Online]. Available: https://developer.arm.com/documentation/ddi0553/bw/.

[26] Y. Wang, J. Wu, T. Yue, Z. Ning, and F. Zhang, « RetTag: Hardware-assisted return address integrity on RISC-V », *in Proceedings of the 15th European Workshop on Systems Security (EUROSEC'22)*, ACM, 2022, pp. 50–56. DOI: 10.1145/3517208.3523758.

[27] S. Weiser, M. Werner, F. Brasser, M. Malenko, S. Mangard, and A.-R. Sadeghi, « TIMBER-V: Tag-isolated memory bringing fine-grained enclaves to RISC-V. », *in Proceedings of the 26th Network and Distributed System Security Symposium (NDSS'19)*, NDSS, 2019. DOI: 10.14722/ndss.2019.23068.

[28] A. Menon, S. Murugan, C. Rebeiro, N. Gala, and K. Veezhinathan, « Shakti-T: a RISC-V processor with light weight security extensions », *in Proceedings of the 6th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP'17)*, ACM, 2017, pp. 1–8. DOI: 10.1145/3092627.3092629.

[29] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, *et al.*, « CHERI: A hybrid capability-system architecture for scalable software compartmentalization », *in Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, IEEE, 2015, pp. 20–37. DOI: 10.1109/sp.2015.9.

[30] R. N. Watson, P. G. Neumann, J. Woodruff, M. Roe, H. Almatary, J. Anderson, J. Baldwin, D. Chisnall, B. Davis, N. W. Filardo, *et al.*, « Capability hardware enhanced RISC instructions: CHERI instruction-set architecture (version 7) », University of Cambridge, Computer Laboratory, Tech. Rep., 2019.

[31] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, « Donky: Domain keys - Efficient In-Process Isolation for RISC-V and x86 », *in Proceedings of the 29th USENIX Security Symposium (Security'20)*, USENIX Association, 2020, pp. 1677–1694. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel.

[32] L. Delshadtehrani, S. Canakci, M. Egele, and A. Joshi, « SealPK: sealable protection keys for RISC-V », *in Design, Automation & Test in Europe Conference & Exhibition (DATE) (DATE'21)*, IEEE, 2021, pp. 1278–1281. DOI: 10.23919/date51398.2021.9473932.

[33] H. Kim, J. Lee, D. Pratama, A. M. Awaludin, H. Kim, and D. Kwon, « RIMI: instruction-level memory isolation for embedded systems on RISC-V », *in Proceedings of the 39th International Conference on Computer-Aided Design (ICCAD'20)*, ACM, 2020, pp. 1–9. DOI: 10.1145/3400302.3415727.

[34] H. Kim, H. T. Larasati, J. Park, H. Kim, and D. Kwon, « DEMIX: Domain-Enforced Memory Isolation for Embedded System », *Sensors*, vol. 23, 7, p. 3568, 2023. DOI: 10.3390/s23073568.

[35] U. Hölzle, C. Chambers, and D. Ungar, « Optimizing dynamically-typed object-oriented languages with polymorphic inline caches », *in Proceedings of the 6th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (ECOOP'91)*, Springer, 1991, pp. 21–38. DOI: 10.1007/bfb0057013.

[36] J. Pallister, S. Hollis, and J. Bennett, « BEEBS: Open benchmarks for energy measurements on embedded platforms », *arXiv Computing Research Repository (CoRR)*, 2013. DOI: 10.48550/arXiv.2205.12742.

[37] S. Singh and M. Awasthi, « Memory centric characterization and analysis of SPEC CPU2017 suite », *in Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE'19)*, ACM, 2019, pp. 285–292. DOI: 10.1145/3297663.3310311.

[38] E. Miranda, C. Béra, E. G. Boix, and D. Ingalls, « Two decades of smalltalk VM development: live VM development through simulation tools », *in Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL'18)*, ACM, 2018, pp. 57–66. DOI: 10.1145/3281287.3281295.

[39] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, *et al.*, « The Rocket chip generator », Tech. Rep., 2016.

[40] N. A. Quynh and D. H. Vu, « Unicorn: Next generation CPU emulator framework », Black Hat USA, 2015, [Online]. Available: https://www.unicorn-engine.org/BHUSA2015-unicorn.pdf.

[41] E. Miranda, « The Cog Smalltalk virtual machine », *in Proceedings of the 5th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL'11)*, ACM, 2011.

[42] G. Polito, P. Tesone, S. Ducasse, L. Fabresse, T. Rogliano, P. Misse-Chanabier, and C. Hernandez Phillips, « Cross-ISA testing of the Pharo VM: lessons learned while porting to ARMv8 », *in Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR'21)*, ACM, 2021, pp. 16–25. DOI: 10.1145/3475738.3480715.

[43] F. Bellard, « QEMU, a Fast and Portable Dynamic Translator », *in Proceedings of the USENIX Annual Conference (ATEC'05)*, USENIX, 2005, pp. 41–46. [Online]. Available: https://www.usenix.org/legacy/events/usenix05/tech/freenix/bellard.html.

[44] M. Williams, M. Tüxen, and R. Seggelmann, *Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension*, 2012. DOI: 10.17487/RFC6520.

[45] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey, *et al.*, « The matter of Heartbleed », *in Proceedings of the 14th ACM SIGCOMM Internet Measurement Conference (IMC'14)*, ACM, 2014, pp. 475–488. DOI: 10.1145/2663716.2663755.

[46] N. Mosier, H. Lachnitt, H. Nemati, and C. Trippel, « Axiomatic hardware-software contracts for security », *in Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA'22)*, ACM, 2022, pp. 72–86. DOI: 10.1145/3470496.3527412.

[47] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, *et al.*, « Meltdown: reading kernel memory from user space », *in Proceedings of the 27th USENIX Security Symposium (Security'18)*, USENIX Association, 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[48] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, *et al.*, « Spectre attacks: exploiting speculative execution », *in Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*, IEEE, 2019, pp. 1–19. DOI: 10.1109/sp.2019.00002.

[49] ECMA. (2023). ECMAScript 2024 Language Specification, [Online]. Available: https://tc39.es/ecma262/.

[50] M. V. Pedersen and A. Askarov, « From trash to treasure: timing-sensitive garbage collection », *in Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P'17)*, IEEE, 2017, pp. 693–709. DOI: 10.1109/sp.2017.64.

[51] D. Everson, L. Cheng, and Z. Zhang, « Log4shell: Redefining the web attack surface », *in Proceedings of the 5th NDSS Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb'22)*, NDSS, 2022. DOI: 10.14722/madweb.2022.23010.

[52] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, « Control-flow integrity principles, implementations, and applications », *TISSEC: ACM Transactions on Information and System Security*, vol. 13, *1*, pp. 1–40, 2009. DOI: 10.1145/1609956.1609960.

[53] A. C. Myers, « JFlow: Practical mostly-static information flow control », *in Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*, ACM, 1999, pp. 228–241. DOI: 10.1145/292540.292561.

[54] M. Castro, M. Costa, and T. Harris, « Securing software by enforcing data-flow integrity », *in Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, USENIX Association, 2006, pp. 147–160. [Online]. Available: https://dl.acm.org/doi/10.5555/1298455.1298470.

[55] ARM, *Learn the architecture - TrustZone for AArch64, Version 1.1*, 2021. [Online]. Available: https://developer.arm.com/documentation/102418/0101/.

[56] A. Waterman, K. Asanovic, and S. Inc, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture*, 2021. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/tag/Priv-v1.12.

[57] J. Smith and R. Nair, *Virtual machines: versatile platforms for systems and processes*. Elsevier, 2005.

[58] A. Goldberg and D. Robson, *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[59] M. Paleczny, C. Vick, and C. Click, « The Java HotSpot server compiler », *in Proceedings of the 1st Java Virtual Machine Research and Technology Symposium (JVM'01)*, USENIX Association, 2001. [Online]. Available: https://www.usenix.org/conference/jvm-01/java-hotspot%E2%84%A2-server-compiler.

[60] D. Ungar and R. B. Smith, « Self: the power of simplicity », *in Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, 1987, pp. 227–242. DOI: 10.1145/38807.38828.

[61] C. Chambers, D. Ungar, and E. Lee, « An efficient implementation of Self a dynamically-typed object-oriented language based on prototypes », *in Proceedings of the 4th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'89)*, ACM, 1989, pp. 49–70. DOI: 10.1145/74877.74884.

[62] U. Hölzle and D. Ungar, « A third-generation Self implementation: reconciling responsiveness with performance », *in Proceedings of the 9th Conference on Object-Oriented Programming Systems, Language, and Applications (OOPSLA'94)*, ACM, 1994, pp. 229–243. DOI: 10.1145/191080.191116.

[63] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson, « Architecture of SOAR: Smalltalk on a RISC », *in Proceedings of the 11th Annual International Symposium on Computer Architecture (ISCA'84)*, ACM, 1984, pp. 188–197. DOI: 10.1145/800015.808182.

[64] W. R. Bush, A. D. Samples, D. Ungar, and P. N. Hilfinger, « Compiling Smalltalk-80 to a RISC », *ACM SIGPLAN Notices*, vol. 22, *10*, pp. 112–116, 1987. DOI: 10.1145/36205.36192.

[65] M. A. Ertl and D. Gregg, « The structure and performance of efficient interpreters », *Journal of Instruction-Level Parallelism (JILP)*, vol. 5, pp. 1–25, 2003. [Online]. Available: https://jilp.org/vol5/v5paper12.pdf.

[66] P. Klint, « Interpretation techniques », *Software: Practice and Experience*, vol. 11, *9*, pp. 963–973, 1981. DOI: 10.1002/spe.4380110908.

[67] R. B. Dewar, « Indirect threaded code », *Communications of the ACM*, vol. 18, *6*, pp. 330–331, 1975. DOI: 10.1145/360825.360849.

[68] E. Rohou, B. N. Swamy, and A. Seznec, « Branch prediction and the performance of interpreters - don't trust folklore », *in Proceedings of the 13th IEEE/ACM International Symposium on Code Generation and Optimization (CGO'15)*, IEEE, 2015, pp. 103–114. DOI: 10.1109/cgo.2015.7054191.

[69] C. F. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo, « Tracing the meta-level: PyPy's tracing JIT compiler », *in Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS'09)*, 2009, pp. 18–25. DOI: 10.1145/1565824.1565827.

[70] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter, « SPUR: a trace-based JIT compiler for CIL », *in Proceedings of the 25th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'10)*, ACM, 2010, pp. 708–725. DOI: 10.1145/1869459.1869517.

[71] C. Chambers and D. Ungar, « Making pure object-oriented languages practical », *in Proceedings of the 6th Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, ACM, 1991, pp. 1–15. DOI: 10.1145/117954.117955.

[72] M. Chevalier-Boisvert, N. Gibbs, J. Boussier, S. X. Wu, A. Patterson, K. Newton, and J. Hawthorn, « YJIT: a basic block versioning JIT compiler for CRuby », *in Proceedings of the 13th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL'21)*, ACM, 2021, pp. 25–32. DOI: 10.1145/3486606.3486781.

[73] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, *et al.*, « Trace-based just-in-time type specialization for dynamic languages », *in Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'09)*, ACM, 2009, pp. 465–478. DOI: 10.1145/1542476.1542528.

[74] U. Hölzle and D. Ungar, « Optimizing Dynamically-Dispatched Calls with Run-Time Type Feedback », *in Proceedings of the 15th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'94)*, ACM, 1994, pp. 326–336. DOI: 10.1145/178243.178478.

[75] R. Jones, A. Hosking, and E. Moss, *The garbage collection handbook: the art of automatic memory management*. CRC Press, 2023. DOI: 10.1201/9781003276142.

[76] D. Gruss, C. Maurice, and S. Mangard, « Rowhammer.js: a remote software-induced fault attack in JavaScript », *in Proceedings of the 13th International conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'16)*, Springer, 2016, pp. 300–321. DOI: 10.1007/978-3-319-40667-1_15.

[77] T. Brennan, N. Rosner, and T. Bultan, « JIT leaks: inducing timing side channels through just-in-time compilation », *in Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, IEEE, 2020, pp. 1207–1222. DOI: 10.1109/sp40000.2020.00007.

[78] T. Brennan, S. Saha, and T. Bultan, « JVM fuzzing for JIT-induced side-channel detection », *in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE'20)*, IEEE, 2020, pp. 1011–1023. DOI: 10.1145/3377811.3380432.

[79] Q. Qin, J. JiYang, F. Song, T. Chen, and X. Xing, « DeJITLeak: eliminating JIT-induced timing side-channel leaks », *in Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'22)*, ACM, 2022, pp. 872–884. DOI: 10.1145/3540250.3549150.

[80] X. Wu, K. Suo, Y. Zhao, and J. Rao, « A side-channel attack on HotSpot heap management », *in Proceedings of the 10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'18)*, USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/hotcloud18/presentation/wu.

[81] A. One, « Smashing the stack for fun and profit », *Phrack magazine*, vol. 7, *49*, 1996. [Online]. Available: http://phrack.org/issues/49/14.html.

[82] A. Prakash and H. Yin, « Defeating ROP through denial of stack pivot », *in Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC'15)*, ACM, 2015, pp. 111–120. DOI: 10.1145/2818000.2818023.

[83] F. Yan, F. Huang, L. Zhao, H. Peng, and Q. Wang, « Baseline is fragile: on the effectiveness of stack pivot defense », *in Proceedings of the 22nd IEEE International Conference on Parallel and Distributed Systems (ICPADS'16)*, IEEE, 2016, pp. 406–413. DOI: 10.1109/icpads.2016.60.

[84] G. Gong, « Pwn a Nexus Device with a Single Vulnerability », CanSecWest, 2016, [Online]. Available: https://github.com/secmob/cansecwest2016.

[85] D. Blazakis, « Interpreter Exploitation », *in Proceedings of the 4th USENIX Workshop on Offensive Technologies (WOOT'10)*, USENIX Association, 2010. [Online]. Available: https://dl.acm.org/doi/10.5555/1925004.1925011.

[86] C. Rohlf and Y. Ivnitskiy, « Attacking clientside JIT compilers », Black Hat USA, 2011, [Online]. Available: https://media.blackhat.com/bh-us-11/Rohlf/BH_US_11_RohlfIvnitskiy_Attacking_Client_Side_JIT_Compilers_WP.pdf.

[87] R. Gawlik and T. Holz, « SoK: make JIT-spray great again », *in Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT'18)*, USENIX Association, 2018. [Online]. Available: https://www.usenix.org/conference/woot18/presentation/gawlik.

[88] H. Shacham, « The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86) », *in Proceedings of the 14th ACM SIGSAC Conference on Computer and Communications Security (CCS'07)*, ACM, 2007, pp. 552–561. DOI: 10.1145/1315245.1315313.

[89] A. Sotirov, « Heap Feng-Shui in JavaScript », Black Hat Europe, 2007, [Online]. Available: https://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf.

[90] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, « G-Free: Defeating Return-Oriented Programming through Gadget-Less Binaries », *in Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC'10)*, IEEE, 2010, pp. 49–58. DOI: 10.1145/1920261.1920269.

[91] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, « Return-oriented programming without returns », *in Proceedings of the 17th ACM SIGSAC Conference on Computer and Communications Security (CCS'10)*, ACM, 2010, pp. 559–572. DOI: 10.1145/1866307.1866370.

[92] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, « Jump-oriented programming: a new class of code-reuse attack », *in Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS'11)*, ACM, 2011, pp. 30–40. DOI: 10.1145/1966913.1966919.

[93] E. Bosman and H. Bos, « Framing signals - A return to portable shellcode », *in Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*, IEEE, 2014, pp. 243–258. DOI: 10.1109/SP.2014.23.

[94] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, « Hacking Blind », *in Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P'14)*, IEEE, 2014, pp. 227–242. DOI: 10.1109/sp.2014.22.

[95] G.-A. Jaloyan, K. Markantonakis, R. N. Akram, D. Robin, K. Mayes, and D. Naccache, « Return-oriented programming on RISC-V », *in Proceedings of the 15th ACM Asia Conference on Computer and Communications Security (ASIACCS'20)*, ACM, 2020, pp. 471–480. DOI: 10.1145/3320269.3384738.

[96] T. Cloosters, D. Paaßen, J. Wang, O. Draissi, P. Jauernig, E. Stapf, L. Davi, and A. Sadeghi, « RiscyROP: automated return-oriented programming attacks on RISC-V and ARM64 », *in Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'22)*, ACM, 2022, pp. 30–42. DOI: 10.1145/3545948.3545997.

[97] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, « Non-control-data attacks are realistic threats », *in Proceedings of the 14th USENIX Security Symposium (Security'05)*, USENIX Association, 2005, pp. 177–191. [Online]. Available: https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats.

[98] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, « Data-oriented programming: on the expressiveness of non-control data attacks », *in Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*, IEEE, 2016, pp. 969–986. DOI: 10.1109/sp.2016.62.

[99] Theori, « Chackra JIT CFG bypass », 2016, [Online]. Available: https://blog.theori.io/research/chakra-jit-cfg-bypass/.

[100] S. Forrest, A. Somayaji, and D. H. Ackley, « Building diverse computer systems », *in Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HotOS'97)*, IEEE, 1997, pp. 67–72. DOI: 10.1109/hotos.1997.595185.

[101] V. Pappas, M. Polychronakis, and A. D. Keromytis, « Smashing the gadgets: Hindering return-oriented programming using in-place code randomization », *in Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P'12)*, IEEE, 2012, pp. 601–615. DOI: 10.1109/SP.2012.41.

[102] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, « Readactor: practical code randomization resilient to memory disclosure », *in Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*, IEEE, 2015, pp. 763–780. DOI: 10.1109/sp.2015.52.

[103] A. Jangda, M. Mishra, and B. De Sutter, « Adaptive just-in-time code diversification », *in Proceedings of the 2nd ACM Workshop on Moving Target Defense (MTD'15)*, ACM, 2015, pp. 49–53. DOI: 10.1145/2808475.2808487.

[104] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, « Address Space Layout Permutation (ASLP): Towards fine-grained randomization of commodity software », *in Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC'06)*, IEEE, 2006, pp. 339–348. DOI: 10.1109/acsac.2006.9.

[105] T. PaX, « PaX team objectives », 2003, [Online]. Available: http://pax.grsecurity.net/docs/pax.txt.

[106] A. Van de Ven, « New security enhancements in Red Hat Enterprise Linux v.3, update 3 », Red Hat, Tech. Rep., 2004.

[107] T. PaX, « PaX Non-Executable pages design (NOEXEC) », 2003, [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt.

[108] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, « You can run but you can't read: preventing disclosure exploits in executable code », *in Proceedings of the 21st ACM SIGSAC Conference on Computer and Communications Security (CCS'14)*, ACM, 2014, pp. 1342–1353. DOI: 10.1145/2660267.2660378.

[109] A. Tang, S. Sethumadhavan, and S. Stolfo, « Heisenbyte: thwarting memory disclosure attacks using destructive code reads », *in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, ACM, 2015, pp. 256–267. DOI: 10.1145/2810103.2813685.

[110] J. Werner, G. Baltas, R. Dallara, N. Otterness, K. Z. Snow, F. Monrose, and M. Polychronakis, « No-Execute-After-Read: preventing code disclosure in commodity software », *in Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIACCS'16)*, ACM, 2016, pp. 35–46. DOI: 10.1145/2897845.2897891.

[111] K. Z. Snow, R. Rogowski, J. Werner, H. Koo, F. Monrose, and M. Polychronakis, « Return to the zombie gadgets: undermining destructive code reads via code inference attacks », *in Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*, IEEE, 2016, pp. 954–968. DOI: 10.1109/sp.2016.61.

[112] I. Fratric, « Bypassing mitigations by attacking JIT server in Microsoft Edge », Google Project Zero, Tech. Rep., 2018. [Online]. Available: https://googleprojectzero.blogspot.com/2018/05/bypassing-mitigations-by-attacking-jit.html.

[113] W. De Groef, N. Nikiforakis, Y. Younan, and F. Piessens, « JITsec: Just-in-Time Security for Code Injection Attacks », *in Workshop on Information and System Security (WISSec'10)*, 2010.

[114] J. Edge, « A seccomp overview », Linux Weekly News, 2015, [Online]. Available: https://lwn.net/Articles/656307/.

[115] N. Burow, X. Zhang, and M. Payer, « SoK: shining light on shadow stacks », *in Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P'19)*, IEEE, 2019, pp. 985–999. DOI: 10.1109/sp.2019.00076.

[116] B. Niu and G. Tan, « Modular control-flow integrity », *in Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, ACM, 2014, pp. 577–587. DOI: 10.1145/2594291.2594295.

[117] M. Schneider, R. J. Masti, S. Shinde, S. Capkun, and R. Perez, « SoK: hardware-supported trusted execution environments », *arXiv Computing Research Repository (CoRR)*, 2022. DOI: 10.48550/arXiv.2205.12742.

[118] V. Costan and S. Devadas, « Intel SGX explained », *Cryptology ePrint Archive (IACR)*, 2016. [Online]. Available: https://eprint.iacr.org/2016/086.

[119] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, « SGAxe: How SGX Fails in Practice », 2020, [Online]. Available: https://sgaxe.com/.

[120] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai, « SmashEx: Smashing SGX Enclaves Using Exceptions », *in Proceedings of the 28th ACM SIGSAC Conference on Computer and Communications Security (CCS'21)*, ACM, 2021, pp. 779–793. DOI: 10.1145/3460120.3484821.

[121] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, « Hodor: process isolation for throughput data plane libraries », *in Proceedings of the USENIX Annual Technical Conference (ATC'19)*, USENIX Association, 2019, pp. 489–504. [Online]. Available: http://www.usenix.org/conference/atc19/presentation/hedayati-hodor.

[122] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, « ERIM: secure, efficient in-process isolation with protection keys (MPK) », *in Proceedings of the 28th USENIX Security Symposium (Security'19)*, USENIX Association, 2019, pp. 1221–1238. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner.

[123] Y. Xu, C. Ye, Y. Solihin, and X. Shen, « Hardware-based domain virtualization for intra-process isolation of persistent memory objects », *in Proceedings of the 47th ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'20)*, IEEE, 2020, pp. 680–692. DOI: 10.1109/isca45697.2020.00062.

[124] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, « libmpk: software abstraction for Intel Memory Protection Keys (Intel MPK) », *in Proceedings of the USENIX Annual Technical Conference (ATC'19)*, USENIX Association, 2019, pp. 241–254. [Online]. Available: https://www.usenix.org/conference/atc19/presentation/park-soyeon.

[125] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee, « Language-independent sandboxing of just-in-time compilation and self-modifying code », *in Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI'11)*, ACM, 2011, pp. 355–366. DOI: 10.1145/1993498.1993540.

[126] L. Delshadtehrani, « Enabling software security mechanisms through architectural support », PhD thesis, Boston University, 2021.

[127] V. Pappas, M. Polychronakis, and A. D. Keromytis, « Transparent ROP exploit mitigation using indirect branch tracing », *in Proceedings of the 22nd USENIX Security Symposium (Security'13)*, USENIX Association, 2013, pp. 447–462. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/paper/pappas.

[128] Y. Cheng, Z. Zhou, Y. Miao, X. Ding, and R. H. Deng, « ROPecker: A generic and practical approach for defending against ROP attack », *in Proceedings of the 21st Network and Distributed System Security Symposium (NDSS'14)*, NDSS, 2014. DOI: 10.14722/ndss.2014.23156.

[129] P. Yuan, Q. Zeng, and X. Ding, « Hardware-assisted fine-grained code-reuse attack detection », *in Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses (RAID'15)*, Springer, 2015, pp. 66–85. DOI: 10.1145/3093336.3037716.

[130] N. Carlini and D. Wagner, « ROP is still dangerous: breaking modern defenses », *in Proceedings of the 23rd USENIX Security Symposium (Security'14)*, USENIX Association, 2014, pp. 385–399. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/carlini.

[131] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz, « Evaluating the effectiveness of current anti-ROP defenses », *in Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'17)*, Springer, 2014, pp. 88–108. DOI: 10.1007/978-3-319-11379-1_5.

[132] R. Mijat, « Better trace for better software: Introducing the new ARM CoreSight system trace macrocell and trace memory controller », Tech. Rep., 2010. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Better_Trace_for_Better_Software_-_CoreSight_STM_with_LTTng_-_19th_October_2010.pdf.

[133] R. Ding, C. Qian, C. Song, B. Harris, T. Kim, and W. Lee, « Efficient protection of path-sensitive control security », *in Proceedings of the 26th USENIX Security Symposium (Security'17)*, USENIX Association, 2017, pp. 131–148. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/ding.

[134] X. Ge, W. Cui, and T. Jaeger, « Griffin: guarding control flows using intel processor trace », *in Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'17)*, ACM, 2017, pp. 585–598. DOI: 10.1145/3037697.3037716.

[135] Y. Gu, Q. Zhao, Y. Zhang, and Z. Lin, « PT-CFI: Transparent backward-edge control-flow violation detection using Intel processor trace », *in Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY'17)*, ACM, 2017, pp. 173–184. DOI: 10.1145/3029806.3029830.

[136] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan, « Transparent and efficient CFI enforcement with intel processor trace », *in Proceedings of the 23rd IEEE International Symposium on High Performance Computer Architecture (HPCA'17)*, IEEE, 2017, pp. 529–540. DOI: 10.1109/hpca.2017.18.

[137] Y. Lee, J. Lee, I. Heo, D. Hwang, and Y. Paek, « Using CoreSight PTM to integrate CRA monitoring IPs in an ARM-Based SoC », *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, *3*, pp. 1–25, 2017. DOI: 10.1145/3035965.

[138] M. A. Wahab, P. Cotret, M. N. Allah, G. Hiet, V. Lapôtre, and G. Gogniat, « ARMHEx: a hardware extension for DIFT on ARM-based SoCs », *in Proceedings of the 27th International Conference on Field Programmable Logic and Applications (FPL'17)*, IEEE, 2017, pp. 1–7.

[139] H. Hu, C. Qian, C. Yagemann, S. P. H. Chung, W. R. Harris, T. Kim, and W. Lee, « Enforcing unique code target property for control-flow integrity », *in Proceedings of the 25th ACM SIGSAC Conference on Computer and Communications Security (CCS'18)*, ACM, 2018, pp. 1470–1486.

[140] Intel, *Intel Trusted Execution Technology (Intel TXT), Revision 017.4*, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/content-details/315168/intel-trusted-execution-technology-intel-txt-software-development-guide.html.

[141] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, « Trusense: Information leakage from trustzone », *in Proceedings of the 37th IEEE Conference on Computer Communications (INFOCOM'18)*, IEEE, 2018, pp. 1097–1105.

[142] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, « SoK: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems », *in Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, IEEE, 2020, pp. 1416–1432. DOI: 10.1109/sp40000.2020.00061.

[143] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, « Keystone: an open framework for architecting trusted execution environments », *in Proceedings of the 15th European Conference on Computer Systems (EuroSys'20)*, ACM, 2020, pp. 1–16. DOI: 10.1145/3342195.3387532.

[144] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, « Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads », *IEEE Transactions on Computers (TC)*, vol. 70, *11*, pp. 1845–1860, 2020. DOI: 10.1109/tc.2020.3027900.

[145] A. Kurth, A. Capotondi, P. Vogel, L. Benini, and A. Marongiu, « HERO: An open-source research platform for HW/SW exploration of heterogeneous many-core systems », *in Proceedings of the 2nd Workshop on AutotuniNg and aDaptivity AppRoaches for Energy efficient HPC Systems (ANDARE'18)*, ACM, 2018, pp. 1–6. DOI: 10.1145/3295816.3295821.

[146] K. Asanovic, D. A. Patterson, and C. Celio, « The berkeley out-of-order machine (BOOM): an industry-competitive, synthesizable, parameterized RISC-V processor », Tech. Rep., 2015.

[147] N. Gala, A. Menon, R. Bodduna, G. Madhusudan, and V. Kamakoti, « SHAKTI processors: an open-source hardware initiative », *in Proceedings of the 29th International Conference on VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID'16)*, IEEE, 2016, pp. 7–8. DOI: 10.1109/vlsid.2016.130.

[148] A. Dörflinger, M. Albers, B. Kleinbeck, Y. Guan, H. Michalik, R. Klink, C. Blochwitz, A. Nechi, and M. Berekovic, « A comparative survey of open-source application-class RISC-V processor implementations », *in Proceedings of the 18th ACM International Conference on Computing Frontiers (CF'21)*, ACM, 2021, pp. 12–20. DOI: 10.24355/dbbs.084-202105101615-0.

[149] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, *et al.*, « SoK:(State of) the art of war: offensive techniques in binary analysis », *in Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P'16)*, IEEE, 2016, pp. 138–157. DOI: 10.1109/sp.2016.17.

[150] O. Gilles, F. Viguier, N. Kosmatov, and D. G. Pérez, « Control-flow integrity at RISC: Attacking RISC-V by jump-oriented programming », *arXiv Computing Research Repository (CoRR)*, 2022. DOI: 10.48550/arXiv.2211.16212.

[151] P. Lab, « 2021 PLCT lab roadmap », 2021, [Online]. Available: https://plctlab.github.io/PLCT-Roadmap-2021.html.

[152] ——, « 2022 PLCT lab roadmap », 2022, [Online]. Available: https://plctlab.github.io/PLCT-Roadmap-2022.html.

[153] J.-E. Ekberg, *Run-Time Protection*, 2021.

[154] T. Lu, « A survey on RISC-V security: Hardware and architecture », *arXiv Computing Research Repository (CoRR)*, 2021. DOI: 10.48550/arXiv.2205.12742.

[155] ARM, *ARM A-Profile Architecture Developments 2022*, 2022. [Online]. Available: https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/arm-a-profile-architecture-2022.

[156] L. Delshadtehrani, S. Canakci, B. Zhou, S. Eldridge, A. Joshi, and M. Egele, « PHMon: a programmable hardware monitor and its security use cases », *in Proceedings of the 29th USENIX Security Symposium (Security'20)*, USENIX Association, 2020, pp. 807–824. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/delshadtehrani.

[157] S. Canakci, L. Delshadtehrani, B. Zhou, A. Joshi, and M. Egele, « Efficient Context-Sensitive CFI Enforcement through a Hardware Monitor », *in Proceedings of the 17th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'20)*, ACM, 2020, pp. 259–279. DOI: 10.1007/978-3-030-52683-2_13.

[158] V. Shanbhogue, « RISC-V CFI Extension », [Online]. Available: https://github.com/riscv/riscv-cfi.

[159] H. Liljestrand, C. Chinea, R. Denis-Courmont, J.-E. Ekberg, and N. Asokan, « Color My World: Deterministic Tagging for Memory Safety », *arXiv Computing Research Repository (CoRR)*, 2022. DOI: 10.48550/2204.03781.

[160] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J.-E. Ekberg, and N. Asokan, « PAC it up: Towards pointer integrity using ARM pointer authentication », *in Proceedings of the 28th USENIX Security Symposium (Security'19)*, USENIX, 2019, pp. 177–194. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/liljestrand.

[161] H. Liljestrand, T. Nyman, L. J. Gunn, J.-E. Ekberg, and N. Asokan, « PACStack: an authenticated call stack », *in Proceedings of the 30th USENIX Security Symposium (Security'21)*, USENIX, 2021, pp. 357–374. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/liljestrand.

[162] J. Li, L. Chen, Q. Xu, L. Tian, G. Shi, K. Chen, and D. Meng, « Zipper stack: Shadow stacks without shadow », *in Proceedings of the 25th European Symposium on Research in Computer Security (ESORICS'20)*, Springer, 2020, pp. 338–358. DOI: 10.1007/978-3-030-58951-6_17.

[163] S. Das, R. H. Unnithan, A. Menon, C. Rebeiro, and K. Veezhinathan, « SHAKTI-MS: a RISC-V processor for memory safety in C », *in Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'19)*, ACM, 2019, pp. 19–32. DOI: 10.1145/3316482.3326356.

[164] L. Delshadtehrani, S. Canakci, W. Blair, M. Egele, and A. Joshi, « FlexFilt: towards flexible instruction filtering for security », *in Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC'21)*, ACM, 2021, pp. 646–659. DOI: 10.1145/3485832.3488019.

[165] M. Xie, C. Wu, Y. Zhang, J. Xu, Y. Lai, Y. Kang, W. Wang, and Z. Wang, « CETIS: Retrofitting Intel CET for generic and efficient intra-process memory isolation », *in Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security (CCS'22)*, ACM, 2022, pp. 2989–3002. DOI: 10.1145/3548606.3559344.

[166] A. Garofalo, M. Rusci, F. Conti, D. Rossi, and L. Benini, « PULP-NN: accelerating quantized neural networks on parallel ultra-low-power RISC-V processors », *Philosophical Transactions of the Royal Society A (RSTA)*, vol. 378, 2020. DOI: 10.1098/rsta.2019.0155.

[167] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, *et al.*, « The gem5 simulator: Version 20.0+ », *arXiv Computing Research Repository (CoRR)*, 2020. DOI: 10.48550/arXiv.2007.03152.

[168] riscv-software-src. (2023). riscv-tests, [Online]. Available: https://github.com/riscv-software-src/riscv-tests.

[169] Veripool. (2023). Verilator, [Online]. Available: https://www.veripool.org/verilator/.

[170] N. A. Quynh, « Capstone: Next-gen disassembly framework », Black Hat USA, 2014, [Online]. Available: https://www.capstone-engine.org/BHUSA2014-capstone.pdf.

[171] S. Marr, B. Daloze, and H. Mössenböck, « Cross-language compiler benchmarking: are we fast yet? », ACM, 2016, pp. 120–131. DOI: 10.1145/2989225.2989232.

[172] K. Shiv, K. Chow, Y. Wang, and D. Petrochenko, « SPECjvm2008 performance characterization », *in Proceedings of the SPEC Benchmark Workshop Computer Performance Evaluation and Benchmarking*, Springer, 2009, pp. 17–35. DOI: 10.1007/978-3-540-93799-9_2.

[173] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, *et al.*, « The DaCapo benchmarks: Java benchmarking development and analysis », *in Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, ACM, 2006, pp. 169–190. DOI: 10.1145/1167473.1167488.

[174] A. WebKit. (2019). JetStream2, a JavaScript and WebAssembly Benchmark suite, [Online]. Available: https://browserbench.org/JetStream/.

[175] Mozilla. (2010). Kraken, a JavaScript performance benchmark suite, [Online]. Available: https://wiki.mozilla.org/Kraken.

[176] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley, « The Jalapeno dynamic optimizing compiler for Java », *in Proceedings of the 1st ACM conference on Java Grande (JAVA'99)*, 1999, pp. 129–141. DOI: 10.1145/304065.304113.

[177] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, « Adaptive optimization in the Jalapeno JVM », *in Proceedings of the 15th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, 2000, pp. 47–65. DOI: 10.1145/353171.353175.

[178] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon, « Maxine: an approachable virtual machine for, and in, Java », *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 9, *4*, pp. 1–24, 2013. DOI: 10.1145/2400682.2400689.

[179] C. Kotselidis, J. Clarkson, A. Rodchenko, A. Nisbet, J. Mawer, and M. Luján, « Heterogeneous managed runtime systems: A computer vision case study », *in Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'17)*, 2017, pp. 74–82. DOI: 10.1145/3050748.3050764.

[180] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay, « Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself », *in Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'97)*, ACM, 1997, pp. 318–326. DOI: 10.1145/263698.263754.

[181] A. Rigo and S. Pedroni, « PyPy's approach to virtual machine construction », *in Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, ACM, 2006, pp. 944–953. DOI: 10.1145/1176617.1176753.

[182] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis, « RPython: a step towards reconciling dynamically and statically typed OO languages », *in Proceedings of the 3rd Symposium on Dynamic languages (DLS'07)*, ACM, 2007, pp. 53–64. DOI: 10.1145/1297081.1297091.

[183] C. Kotselidis, A. Nisbet, F. S. Zakkak, and N. Foutris, « Cross-ISA debugging in meta-circular VMs », *in Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL'17)*, ACM, 2017, pp. 1–9. DOI: 10.1145/3141871.3141872.

[184] D. Patterson and A. Waterman, *The RISC-V Reader: an open architecture Atlas*. Strawberry Canyon, 2017.

[185] M. Maas, A. Zabrocki, and C. Stephano, « RISC-V J Extension », [Online]. Available: https://github.com/riscv/riscv-j-extension.

[186] T. PaX, « PaX Address Space Layout Randomization (ASLR) », 2003, [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt.

[187] C. Curtsinger and E. D. Berger, « Stabilizer: statistically sound performance evaluation », *ACM Computer Architecture News (SIGARCH)*, vol. 41, *1*, pp. 219–228, 2013. DOI: 10.1145/2490301.2451141.

[188] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi, « Timely rerandomization for mitigating memory disclosures », *in Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*, ACM, 2015, pp. 268–279. DOI: 10.1145/2810103.2813691.

[189] Y. Chen, Z. Wang, D. Whalley, and L. Lu, « Remix: on-demand live randomization », *in Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16)*, ACM, 2016, pp. 50–61. DOI: 10.1145/2857705.2857726.

[190] K. Lu, W. Lee, S. Nürnberger, and M. Backes, « How to make ASLR win the clone wars: runtime re-randomization », *in Proceedings of the 23rd Network and Distributed System Security Symposium (NDSS'16)*, NDSS, 2016. DOI: 10.14722/ndss.2016.23173.

[191] D. Williams-King, G. Gobieski, K. Williams-King, J. P. Blake, X. Yuan, P. Colp, M. Zheng, V. P. Kemerlis, J. Yang, and W. Aiello, « Shuffler: fast and deployable continuous code re-randomization », *in Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*, ACM, 2016, pp. 367–382. [Online]. Available: https://dl.acm.org/doi/10.5555/3026877.3026906.

[192] X. Chen, H. Bos, and C. Giuffrida, « CodeArmor: virtualizing the code space to counter disclosure attacks », *in Proceedings of the 2nd IEEE European Symposium on Security and Privacy (EuroS&P'17)*, IEEE, 2017, pp. 514–529. DOI: 10.1109/eurosp.2017.17.

[193] S. Ahmed, Y. Xiao, K. Z. Snow, G. Tan, F. Monrose, and D. Yao, « Methodologies for quantifying (re-)randomization security and timing under JIT-ROP », *in Proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security (CCS'20)*, ACM, 2020, pp. 1803–1820. DOI: 10.1145/3372297.3417248.

[194] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose, « Isomeron: code randomization resilient to (just-in-time) return-oriented programming », *in Proceedings of the 22nd Network and Distributed System Security Symposium (NDSS'15)*, NDSS, 2015. DOI: 10.14722/ndss.2015.23262.

[195] M. Backes and S. Nürnberger, « Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing », *in Proceedings of the 23rd USENIX Security Symposium (Security'14)*, USENIX Association, 2014, pp. 433–447. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/backes.

[196] T. Garfinkel, B. Pfaff, M. Rosenblum, *et al.*, « Ostia: a delegating architecture for secure system call interposition », *in Proceedings of the 11th Network and Distributed System Security Symposium (NDSS'04)*, NDSS, 2004. [Online]. Available: https://www.ndss-symposium.org/ndss2004/ostia-delegating-architecture-secure-system-call-interposition/.

[197] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, « Native Client: a sandbox for portable, untrusted x86 native code », *Communications of the ACM*, vol. 53, *1*, pp. 91–99, 2010. DOI: 10.1145/1629175.1629203.

COLLEGES SCIENCES
BRETAGNE POUR L'INGENIEUR
LOIRE ET LE NUMERIQUE

ENSTA BRETAGNE

Titre : Sécurisation matérielle de la compilation à la volée de machines virtuelles langage

Mot clés : Machine virtuelle (VM), Compilateur à la volée (JIT), RISC-V, Isolation matérielle

Résumé : Les machines virtuelles langage (VM) sont l'environnement d'exécution des langages de haut niveau les plus répandus. Elles permettent une portabilité du code applicatif et la gestion automatique de la mémoire. Leur large diffusion couplée à l'exécution de tâches de bas niveau les rendent intéressantes pour les attaquants. Les solutions purement logicielles entraînent souvent une perte de performance incompatible avec la compilation *just-in-time (JIT)*. Des solutions accélérées matériellement sont ajoutées dans des processeurs commerciaux pour concilier des garanties de sécurité fortes avec la performance. Pour comparer ces solutions, cette thèse s'intéresse au jeu d'instructions RISC-V et à ses capacités d'extension. Nous présentons Gigue, un générateur de binaires similaires au code JIT directement exécutables sur les softcores RISC-V. Il fournit une interface pour des instructions personnalisées et garantit leur exécution. Nous présentons une solution d'isolation de domaine au niveau des instructions ajoutée aux binaires de Gigue et déployée dans un processeur avec des modifications minimales. La solution ajoute un surcoût de performance négligeable tout en garantissant des propriétés fortes sur les domaines. Afin de motiver le déploiement dans des cas d'utilisation réels, nous étendons le compilateur JIT Pharo au jeu d'instructions RISC-V, ainsi que son infrastructure de test.

Title: Hardware security for just-in-time compilation in language virtual machines

Keywords: Virtual Machines (VM), Just-in-time (JIT) compiler, RISC-V, Hardware isolation

Abstract: Language Virtual Machines (VMs) are the run-time environment of popular high-level managed languages. They offer portability and memory handling for the developer and are deployed on most computing devices. Their widespread distribution, handling of untrusted user inputs, and low-level task execution make them interesting to attackers. Software-only solutions that isolate their different components often incur a high performance overhead incompatible with *just-in-time (JIT)* compilation. Hardware-accelerated runtime protections are pushed in vendor processors as a solution to conciliate strong security guarantees with performance. To allow experimentation in the design and comparison of such solutions, this thesis is interested in the RISC-V instruction set and its extension capabilities. We present Gigue, a workload generator that outputs binaries similar to JIT code directly executable on RISC-V softcores. It provides an interface for custom instructions and guarantees their execution. We present an instruction-level domain isolation solution added to Gigue binaries and implemented in an application-class processor with processor modifications. The solution adds negligible performance overhead while enforcing strong properties on domains. As an effort to motivate deployment in real use cases, we extend the Pharo JIT compiler to the RISC-V instruction set along with its testing infrastructure.