COLLEGE DOCTORAL BRETAGNE
MATHS, TELECOMS
INFORMATIQUE, SIGNAL
SYSTEMES, ELECTRONIQUE

Université de Rennes

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601
Mathématiques, Télécommunications, Informatique, Signal, Systèmes, Électronique
Spécialité : *Informatique*

Par
Pierrick PHILIPPE

Secrets in Compiler: Detection of Secret-related Weaknesses in GCC Static Analyzer

Secrets dans le Compilateur: Détection de Faiblesses Liées au Secret dans l'Analyseur Statique de GCC

Thèse présentée et soutenue à Rennes, le 10 décembre 2025
Unité de recherche : UMR 6074

Rapporteur·trice·s avant soutenance :

Benjamin GRÉGOIRE — Directeur de Recherche, Inria Sophia-Antipolis, France
Christophe HAUSER — Assistant Proffesor, Darthmouth College, USA

Composition du Jury :

Président·e :	Isabelle PUAUT	Professeure, Université de Rennes, France
Examinateur·ice·s :	Lesly-Ann DANIEL	Assistant Professor, EURECOM, France
	Alessandro MANTOVANI	Security Researcher, QUALCOMM, USA
	Benjamin GRÉGOIRE	Directeur de Recherche, Inria Sophia-Antipolis, France
	Christophe HAUSER	Assistant Proffesor, Darthmouth College, USA
Direction :	Pierre-Alain FOUQUE	Professeur, Université de Rennes, France
Encadrement :	Mohamed SABT	Professeur, Université de Rennes, France

Remerciements

Only in printed versions.

Résumé en Français

La cybersécurité fait partie intégrante de notre vie quotidienne, qu'il s'agisse d'appareils personnels ou de services nationaux essentiels ; les systèmes logiciels sont omniprésents. Par conséquent, tout système vulnérable peut avoir de graves conséquences, allant d'atteintes à la vie privée jusqu'à des perturbations globales de la société [1–3]. De ce fait, la fiabilité et la sécurité des systèmes logiciels influencent directement la confiance des utilisateurs et des organisations accordée à ces systèmes.

Les facteurs qui menacent la sécurité des logiciels sont particulièrement répandus, entre erreurs de programmation et défauts de conception. Ils conduisent souvent à des faiblesses que des attaquants peuvent exploiter pour compromettre la confidentialité, l'intégrité ou la disponibilité d'un système. Parmi ces défauts de programmation, les erreurs de gestion de la mémoire sont très fréquentes [4]. Elles sont particulièrement dangereuses lorsqu'elle concerne des données sensible, car elle peuvent être divulguées à des acteurs non autorisés, ce qui a pour conséquence de réduire la sécurité, comme l'illustre l'attaque Heartbleed [5]. Veiller à ce que les données sensibles soient correctement gérées est donc une exigence fondamentale pour assurer la fiabilité et la sécurité des systèmes d'information.

Dans ce contexte, les questions liées à la gestion des données sensibles méritent une attention accrue. En particulier, les secrets ne devraient pas persister en mémoire au-delà de leur durée de vie prévue, comme le suggèrent les *SEI CERT C Coding Standards* [6]. Ne pas appliquer ces règles ne constitue pas seulement une violation des bonnes pratiques, mais laisse également les systèmes vulnérables à la fuite d'informations.

La société MITRE gère la base de données Common Weaknesses Enumeration (CWEs) qui fournit un catalogue de ces failles logicielles visant à identifier les causes sous-jacentes de Common Vulnerabilities and Exposures (CVE). Deux CWEs ciblent l'incapacité à supprimer correctement les données sensibles en mémoire, ce qui peut conduire à leur exposition à des acteurs non autorisés : CWE-226 (*Sensitive Information in Resource Not Removed Before Reuse*) [7] et CWE-244 (*Improper Clearing of Heap Memory Before Release*) [8]. Récemment, la vulnérabilité CVE-2025-1759 [9] visant IBM Concert permet à un attaquant distant de récupérer des données sensibles non nettoyées sur le tas, en lien avec la CWE-244. D'autres CWEs décrivent des failles divulguant des données sensibles dans un contexte de traitement d'erreur pour la CWE-209 (*Generation of Error Message Containing Sensitive Information*) [10] ou à travers le comportement du système pour la CWE-203 (*Observable Discrepancy*) [11]. La récente CVE-2025-49128 [12] ciblant une bibliothèque d'analyse JSON conduit à une divulgation potentielle d'informations et est donc liée à la CWE-209. Ces deux faiblesses relèvent de la CWE-200 (*Exposure of Sensitive Information to an Unauthorized Actor*), qui englobe les divulgations involontaires de données sensibles [13]. Ces récentes vulnérabilités démontrent qu'une mauvaise gestion des secrets n'est pas une préoccupation théorique ou dépassée, mais qu'elle reste une source réelle de vulnérabilités qui appelle à une solution automatique pour leur détection.

Un moyen courant de détecter ces faiblesses est de s'appuyer sur l'analyse de programme, qui fournit une approche systématique pour raisonner sur les propriétés fonctionnelles et non fonctionnelles d'un programme. Les différences entre les deux sont relativement simples à comprendre. Les propriétés fonctionnelles d'un programme indiquent son ou ses objectifs. En revanche, ses propriétés non fonctionnelles mettent en évidence tout le reste, y compris, mais sans s'y limiter, la sûreté, la sécurité, la facilité d'utilisation, la portabilité et la maintenabilité du programme, ainsi que des informations sur sa licence (par exemple, en source ouverte ou fermée).
En examinant les programmes, il devient possible de trouver des comportements qui pourraient enfreindre ces propriétés, ce qui conduit généralement à des vulnérabilités. Ces techniques sont devenues essentielles dans l'ingénierie logicielle moderne, car l'examen manuel n'est pas toujours fiable, en particulier pour les grandes bases de code.

Les techniques d'analyse de programme sont généralement divisées en trois catégories :

1. L'**analyse statique** inspecte le code en raisonnant sur ses propriétés, mais elle nécessite des approximations, ce qui peut conduire à des émissions de faux-positif.

2. Les techniques d'**analyse dynamique** observent le comportement du programme durant son exécution et peuvent détecter des vulnérabilités concrètes, bien qu'elles soient limitées aux chemins réellement exécutés et qu'elles s'accompagnent généralement d'une augmentation du temps d'exécution.
3. Les techniques d'**analyses hybrides** tentent de combiner les méthodes statiques et dynamiques, en tirant parti des points forts de chacune d'elles pour obtenir une plus grande précision et une meilleure couverture du code.

Les compilateurs sont une classe spécifique de programmes utilisant l'analyse de programme et occupant une position clé dans le cycle de développement des logiciels. En bref, ils traduisent un code source de haut niveau compréhensible pour les humains en code machine bas niveau. Cependant, ils ne se contentent généralement pas de traduire le code, mais l'optimisent également grâce à un large éventail de techniques d'analyse statique issues de la théorie des compilateurs, comme l'illustre le *Dragon Book* de A. V. Aho *et al.* [14].

GCC est l'un des compilateurs les plus utilisés, en particulier pour le code C, selon une enquête de 2023 sur l'écosystème des développeurs de JetBrains [15], prenant en charge de nombreux langages de programmation et architectures. Au sein de GCC, son analyseur statique fournit un moteur d'exécution symbolique capable d'explorer les différents chemins des programmes C depuis la sorite de GCC 10 [16]. En résumé, le GCC Static Analyzer (GSA) est un cadriciel d'analyse statique abstrayant les structures de données internes de GCC et tirant parti d'un moteur d'exécution symbolique et de différentes structures de données pour mettre en œuvre un large éventail d'analyses, permettant ainsi de détecter des vulnérabilités au moment de la compilation [17].

Cette thèse se concentre sur l'analyse statique directement au sein du compilateur GCC en étendant son analyseur statique pour détecter les faiblesses liées au secret mentionnées ci-dessus. En disséquant son architecture et en étendant ses capacités, nous visons à adapter le GSA au problème des faiblesses liées au secret. Nos contributions démontrent comment l'analyse intégrée au compilateur peut à la fois détecter une réduction à zéro manquante dans la pratique, à savoir CWE-226 et CWE-244, ce qui peut ensuite être généralisé à des catégories plus larges de fuites d'informations sensibles, c.-à-d., descendant de CWE-200. Grâce à nos travaux, le GSA s'avère être non seulement un outil généraliste de détection des bogues, mais aussi un cadriciel prometteur pour améliorer la sécurité des logiciels.

Le choix du GSA est motivé par plusieurs aspects :

1. Les développeurs ont tendance à réprouver les outils qui interrompent leur méthodes de travail, selon diverses enquêtes [18–20]. Cependant, les compilateurs font déjà partie des méthodes et du cycle de vie du développement logiciel, ce qui en fait des candidats naturels pour les techniques d'analyse statique visant à améliorer la sécurité des logiciels à grande échelle.
2. L'architecture monolithique de GCC lui permet d'être responsable de chaque transformation de code, sans dépendre de l'analyse d'autres programmes. Contrairement à la relation Clang/LLVM, Clang s'appuie principalement sur les transformations de code fournies par le compilateur LLVM plutôt qu'au sein de Clang directement.
3. Le GSA est placé à un endroit intéressant dans le bitoduc de compilation de GCC, c'est-à-dire à la dernière étape avant la fin des passes indépendantes du langage et de la machine. Jusqu'à présent, il ne prend en charge que l'analyse du code C, bien que des travaux soient en cours pour l'étendre à C++ pour la version 16 de GCC. Cependant, il reste un cadriciel d'analyse statique prometteur intégré directement dans le compilateur initial du noyau GNU/Linux.

1. Contributions

Dans cette thèse, nous étudions comment l'analyse statique intégrée au compilateur peut être étendue pour traiter les vulnérabilités liées au secret. Nous abordons cette investigation à travers trois questions de recherche, chacune guidant l'une de nos principales contributions :

I. *Dans quelle mesure le GSA convient-il pour les analyses liées à la sécurité ?*

II. *Une analyse axée sur un cas d'utilisation de la fuite d'informations peut-elle être mise en œuvre qu sein du GSA ?*

III. *Comment une analyse basée sur le GSA peut-elle être généralisée pour détecter un spectre plus large de faiblesses liées à des fuites d'information ?*

Les contributions de cette thèse sont divisées en trois catégories complémentaires. Premièrement, nous disséquons le GSA pour évaluer son architecture et ses éléments internes, et ainsi déterminer s'il convient pour des analyses de sécurité. Deuxièmement, nous étendons l'analyseur avec GnuZero, un outil pour détecter l'abscence de mise à

zéro des données sensibles dans du code C, en se concentrant sur les CWE-226 et CWE-244. Cette contribution introduit de nouveaux attributs de langage et un modèle de propagation de teinte, exploite le système de diagnostique expressif du GSA et est validée à la fois sur des bancs d'essai et des projets logiciels en source ouverte. Enfin, nous proposons GnuSecret comme une généralisation de l'approche de GnuZero, en unifiant le traitement des faiblesses liées aux secrets sous un nouvel ensemble d'attributs du language C. En propageant les secrets à travers les variables du programme et en détectant les flux non sécurisés vers différents puits, GnuSecret démontre sa capacité à capturer des vulnérabilités plus larges telles que celles relevant du champ d'application de la CWE-200, en repérant à la fois des CVEs connues et de nouvelles dans des logiciels complexes. Ensemble, ces contributions illustrent la faisabilité de l'intégration d'une analyse de sécurité directement dans un compilateur très utilisé.

1.1. Dissection du GSA

La première contribution de cette thèse est la fondation des contributions suivantes, et consiste en une dissection en profondeur du GCC Static Analyzer (GSA). Cette plongée dans ses méchanismes internes clarifie son moteur d'exécution symbolique, son modèle de mémoire, son système de diagnostique, et son architecture globale. Grâce à cela, nous établissons les bases sur lesquelles de nouveaux mécanismes de détection de vulnérabilité peuvent être construits de manière fiable, tout en mettant en évidence les avantages et les limites du GSA. Dans l'ensemble, cette contribution vise à répondre à la question **I** et motive une extension du GSA pour les cas d'utilisation de fuite d'information.

Dans le cadre de cette dissection, nous décrivons le supergraphe éclaté que le GSA utilise pour l'analyse du flux de contrôle interprocédural, ainsi que son modèle de mémoire basé sur les régions, qui est utilisé pour représenter la pile et le tas. Nous examinons également le système de diagnostique GSA, en montrant comment les avertissements sont construits et tracés jusqu'aux emplacements dans le fichier source, ainsi que l'interaction entre le système d'attributs GNU et le GSA, en identifiant comment les attributs peuvent être utilisés pour étendre la sémantique du langage afin de conduire l'analyse.

Pour évaluer l'efficacité du GSA sur des bases de code réelles, nous l'avons évalué sur un sous-ensemble de la suite de test Juliet [21], en particulier le sous-ensemble ciblant à la fois CWE-214 (*Double Free*) et CWE-215 (*Use-After-Free*). Nous reproduisons également la détection de CVE-2020-1967 dans OpenSSL [22], initialement détectée par le GSA.

Pour résumer cette contribution, nous documentons et détaillons les éléments internes du GSA, créant ainsi une documentation technique centralisée issue de la pluralité des documentations du projet GCC. C'est le travail fondateur des contributions suivantes.

1.2. GnuZero : Un outil statique pour détecter la non-zéroisation

La mise à zéro est la pratique qui consiste à effacer explicitement les informations sensibles de la mémoire une fois qu'elles ne sont plus nécessaires. Il s'agit d'une faiblesse liée à la fuite d'informations, qui est le cas d'utilisation sélectionné pour répondre à la question **II**. L'objectif est de s'assurer que les secrets ne persistent pas dans la mémoire du programme au-delà de leur durée de vie prévue, ce qui permettrait à des attaquants de les récupérer plus tard. Naïvement, le problème est simple : il suffit d'effacer les données sensibles. Malheureusement, la mise à zéro est difficile à appliquer en pratique [23]. Même lorsque les développeurs appellent des fonctions de nettoyage explicites, les optimisations du compilateur peuvent les supprimer silencieusement. En effet, les compilateurs qui optimisent de manière agressive peuvent détecter des opérations d'écriture sur des objets en mémoire qui ne seront pas lus par la suite. C'est ce qu'on appelle le problème de *Dead-Store Elimination*. En outre, les secrets se propagent à travers les variables temporaires insérés par le compilateur ou les paramètres de fonction, ce qui rend pénible pour les développeurs le suivi de tous les objets de mémoire qui nécessitent une remise à zéro.

Basé sur la conclusion de la dissection du GSA, notre deuxième contribution est GnuZero, un outil basé sur le compilateur pour détecter l'abscence de mise à zéro des données sensibles. Ciblant les CWE-226 et CWE-244, GnuZero étend le système d'attributs GNU pour permettre aux développeurs d'assister notre analyse. Nous introduisons deux attributs : `scrub` pour marquer les variables sensibles et `scrubber` pour identifier les fonctions de mise à zéro. En outre, nous définissons un modèle de propagation de teinte pour identifier automatiquement les variables de la pile et du tas non nettoyées à la fin de leur durée de vie. Notre évaluation montre que GnuZero atteint une couverture complète sur un banc d'essai dédié à CWEs, à savoir la suite de test Juliet [21]. Nous évaluons également GnuZero sur des CVEs connues et détectons de nouvelles vulnérabilités de fuite d'information dans des projets en source ouverte.

Reference: Ces travaux sont le résultat d'une collaboration avec Mohamed Sabt et Pierre-Alain Fouque. Ils ont été publiés dans les actes de la *IEEE/IFIP International Conference on Dependable Systems and Networks 2025* [24], et a reçu l'un des trois *Best Papers Awards*, ainsi que le *Distinguished Paper Award*.

1.3. GnuSecret

Notre première contribution montre que le GSA convient bien à une analyse ciblant à la fois CWE-226 et CWE-244 directement dans le compilateur GCC. Cependant, le spectre des faiblesses liées à la divulgation d'informations sensibles est plus large, comme l'illustrent les nombreux enfants de CWE-200 [13]. Il existe plusieurs outils issus de la communauté de la recherche qui ciblent différentes vulnérabilités liées à la fuite d'informations, en dépit de leurs similitudes. Pire encore, chacun d'entre eux a son propre système d'annotation, son propre système de diagnostique et ses propres techniques d'analyse statique [25–29], augmentant la confusion parmi les développeurs concernés [30].

Notre dernière contribution, appelée GnuSecret, est un cadre qui généralise l'approche de GnuZero à des vulnérabilités de divulgation d'informations plus larges et vise à répondre à la question de recherche **III**. En introduisant plusieurs attributs liés au secret, il unifie la sémantique du suivi du secret à travers les variables, les fonctions et les types C. Combiné au moteur d'exécution symbolique de l'analyseur, GnuSecret propage les secrets sur la base du flux de données du programme et détecte les flux non sécurisés vers des puits permettant une fuite de ces données. Pour évaluer GnuSecret, nous avons analysé des projets en source ouverte connus pour être vulnérables afin d'évaluer l'efficacité de GnuSecret : GnuTLS [31], OpenSC [32], et libJWT [33]. Enfin, GnuSecret a détecté une vulnérabilité inconnue dans un module cryptographique du noyau GNU/linux. L'analyse d'un code vulnérable connu et la détection d'une nouvelle vulnérabilité illustrent l'efficacité de l'approche de GnuSecret en pratique.

Reference: Ce travail est le résultat d'une collaboration avec Mohamed Sabt, Abdoulaye Katchala Mele, et Pierre-Alain Fouque. Il fait l'objet d'une soumission en cours dans une conférence académique.

2. Artéfacts

Dans un effort pour promouvoir la science ouverte, toutes nos contributions présentent des artefacts accessibles au public sur Zenodo et des dépôts Git sur l'instance GitLab de l'INRIA. Une archive a été publiée sur Zenodo [34], contenant tous les artefacts de cette thèse, y compris les artefacts de l'évaluation du GSA, de GnuZero et GnuSecret.

GCC modifié: basé sur un commit spécifique [35], le code source est disponible à l'addresse suivante : https://gitlab.inria.fr/pphilipp/gcc/-/tree/analyzer-tracking-region?ref_type=heads. Nos modifications ont été présentées à la conférence technique FOSDEM en 2024 [36] auprès de la communauté des développeurs de GCC [37].

Évaluation du GSA: introduit dans la Section 1.1. Une archive est disponible sur Zenodo [38], contenant :

1. les sources de la suite de tests Juliet pour les CWE-214 et CWE-215, ainsi que les fichiers SARIF de l'analyse et un script Python pour exécuter les tests,
2. un dépôt cloné d'OpenSSL version 1.1.1f, avec un script pour reproduire l'analyse qui a conduit à la détection de CVE-2020-1967 [22].

GnuZero: introduit dans la Section 1.2. Le code source de GnuZero est disponible à l'adresse suivante : https://gitlab.inria.fr/pphilipp/plugin-scrub. En plus du code source, une archive spécifique à l'évaluation de GnuZero est disponible sur Zenodo [39] contenant :

1. la version modifiée de GCC au moment de la publication,
2. des utilitaires pour construire à partir des sources un paquet Debian contenant la version modifiée compilée de GCC (bien qu'un paquet déjà construit soit présent),
3. les sources adaptées de la suite de tests Juliet pour les CWE-226 et CWE-244, ainsi que les fichiers SARIF de l'analyse et un script Python pour exécuter les tests,
4. et le code source de GnuZero et des différents tests.

GnuSecret: introduit dans la Section 1.2. Le code source de GnuSecret est disponible à l'adresse suivante : https://gitlab.inria.fr/pphilipp/gcc-plugin-ct. En plus du code source, une archive spécifique à l'évaluation de GnuSecret est disponible sur Zenodo [40] contenant :

1. une version modifiée du compilateur GNU Compiler Collection (GCC),

2. des utilitaires pour construire à partir des sources un paquet Debian contenant la version modifiée compilée de GCC (bien qu'un paquet déjà construit soit présent),
3. le code source adapté des bibliothèques analysées, accompagné de fichiers SARIF et d'un script Python pour l'exécution des différents tests,
4. et le code source de GnuSecret.

PUBLICATIONS

In this chapter, we list peer-reviewed publications in the proceedings of international conferences in chronological order and ongoing submissions in such conferences. This thesis is built on the basis of these publications and submissions.

Publication I	[24]	"GnuZero: A Compiler-Based Zeroization Static Detection Tool for the Masses" Pierrick Philippe, Mohamed Sabt, Pierre-Alain Fouque. Published in the proceedings of DSN 2025 (pp. 208-221)
Ongoing Submission		"Tentative Definition of the Secret Attribute in the C Language" Pierrick Philippe, Mohamed Sabt, Abdoulaye Katchala-Mele, Pierre-Alain Fouque.

Annotations and Abbreviations

AST – Abstract Syntax Tree

CFG – Control-Flow Graph

CI – Continuous Integration

CLI – Command Line Interface

CVE – Common Vulnerabilities and Exposures

CWE – Common Weakness Enumeration

DAG – Directed Acyclic Graph

DFA – Data-Flow Analysis

DSA – Data Structure Analysis

DSE – Dead-Store Elimination

EH – Exception Handler

GCC – GNU Compiler Collection

GSA – GCC Static Analyzer

IDE – Integrated Development Environment

IFDS – *Interprocedural, Finite, Distributive, Subset*

IL – Intermediate Language

IR – Intermediate Representation

LHS – left-hand side

MD – Machine Description

NPD – Null Pointer Dereference

OMP – OpenMP

OOB – Out-Of-Bounds

RHS – right-hand side

RTL – Register Transfer Language

SARIF – Static Analysis Results Interchange Format

SSA – Static Single-Assignment

UAF – Use-After Free

UB – Undefined Behavior

TABLE OF CONTENTS

Remerciements ..... **v**

Résumé en Français ..... **vii**
1. Contributions ..... *viii*
2. Artéfacts ..... *x*

Publications ..... **xiii**

Annotations and Abbreviations ..... **xv**

Table of contents ..... **xvii**

1.1. Introduction **19**
1.1. Contribution of this Thesis *20*
1.2. Artifacts *22*
1.3. Outline *22*

I Preliminaries **25**

2.1. GCC: The GNU Compiler Collection **27**
2.1. A Brief History of GCC *29*
2.2. GCC Internal Representations *29*
2.3. GNU Attribute System *36*
2.4. GCC's Warning Engine Limitations *38*
2.5. Plugins *38*

3.2. Program Analysis **41**
3.1. Soundness vs. Completeness *42*
3.2. Static Analysis *42*
3.3. Dynamic Analysis *47*
3.4. Hybrid Analysis *48*

II Contributions **49**

4.1. GCC Static Analyzer **51**
4.1. Context and Motivations *52*
4.2. Exploded Supergraph *52*
4.3. Region-based Memory Model *53*
4.4. Program State *57*
4.5. State Machine *58*
4.6. Reporting System *61*
4.7. GNU Attributes System and GSA Relationship *62*
4.8. Native GSA's warning *63*
4.9. GSA vs. the World *66*
4.10. Conclusion *67*

5.2. GnuZero: A Compiler-Based Zeroization Static Detection Tool for the Masses **69**
5.1. Context and Motivations *70*
5.2. Enhancing the GSA *71*
5.3. Zeroization Analysis *72*
5.4. The Scrub Attributes *74*
5.5. Zeroization with Tainting Analysis *76*
5.6. Zeroization State Machine *79*

5.7. Evaluation *79*
5.8. Related Work *83*
5.9. Conclusion *84*

6.3. Tentative Definition of the Secret Attribute in the C Language **87**
6.1. Context and Motivations *88*
6.2. The Secret Attribute *89*
6.3. The Secret Propagation *93*
6.4. Sinks for Information Disclosure *95*
6.5. Evaluation *97*
6.6. Related Work *98*
6.7. Conclusion *100*

III Conclusion and Future Work **101**

7.1. Conclusion **103**
7.1. Summary of Contributions *104*
7.2. Limitations *104*
7.3. Future Works *105*
7.4. The Big Picture *105*
7.5. Discussion: to Warn, or to Lint, that is the Question *106*

Bibliography **107**

Chapter 1

Introduction

Cybersecurity is integrated into our daily lives, from personal devices to critical national services; software systems are ubiquitous. Hence, any vulnerable system can lead to severe consequences, spanning from privacy breaches to societal disturbance [1–3]. The reliability and security of software systems, therefore, directly influence users' and organizations' trust in such systems.

Among the factors endangering software security, programming errors and design flaws emerge as particularly pervasive, often leading to exploitable weaknesses that attackers can leverage to compromise a system's confidentiality, integrity, or availability. Amidst those programming flaws, memory management errors are highly common [4]. They are especially dangerous concerning sensitive data, as they could be leaked to unauthorized actors, therefore reducing security, as illustrated by the Heartbleed attack [5]. Ensuring that sensitive data is correctly managed is thus a fundamental requirement to reach both reliability and security for information systems.

In this context, issues related to secret data management deserve singular attention. Specifically, secrets should not persist in memory beyond their intended lifetime, as suggested by the *SEI CERT C Coding Standards* [6]. Failure to enforce it not only violates best practices but also leaves systems vulnerable to information disclosure. Consequently, memory safety and secret management remain vital for state-of-the-art software security engineering. Alas, the notion of sensitivity is context and application-dependent: (1) cryptographic keys for cryptographic libraries, (2) passwords for authentication code, or even (3) addresses of specific memory objects when considering the GNU/Linux kernel.

The MITRE corporation maintains the Common Weaknesses Enumeration (CWEs) database that provides a catalog of such software flaws that aims to identify the underlying causes of Common Vulnerabilities and Exposures (CVE). Two CWEs target the failure to properly remove sensitive data from memory, potentially leading to data exposure to unauthorized actors: CWE-226 (*Sensitive Information in Resource Not Removed Before Reuse*) [7] and CWE-244 (*Improper Clearing of Heap Memory Before Release*) [8]. Recently, the vulnerability labeled CVE-2025-1759 [9] in IBM Concert exposed sensitive information to a remote attacker due to improper heap clearing, classified under CWE-244. Other CWEs describe flaws leaking either sensitive data in an error-handling context for CWE-209 (*Generation of Error Message Containing Sensitive Information*) [10] or through the system's behavior for CWE-203 (*Observable Discrepancy*) [11]. The recent CVE-2025-49128 [12] in a JSON parsing library was leading to potential information disclosure and classified under CWE-209. These two weaknesses fall under CWE-200 (*Exposure of Sensitive Information to an Unauthorized Actor*), which captures unintended sensitive data disclosures [13]. Such cases show that poor secret management is not a theoretical or outdated concern but still an actual source of vulnerabilities and motivates a solution for automatic detection of those flaws.

A common way to detect these weaknesses is to rely on program analysis, which provides a systematic approach to reason about functional and non-functional properties of a program. The differences between the two are relatively simple to grasp. Functional properties of a program indicate its purpose(s). In contrast, its non-functional properties highlight everything else, including, but not limited to, the program's safety, security, usability, portability, and maintainability, as well as information about its licensing (e.g., open or closed source).
By examining programs, it becomes possible to find behavior that might break those properties, usually leading to vulnerabilities. Such techniques have become essential in modern software engineering, as manual review is not always reliable, especially for large codebases.

Program analysis techniques are commonly divided into three categories:

1. **Static analysis** inspects code by reasoning on the code's properties, but it requires approximations, potentially leading to false positive emissions.
2. **Dynamic analysis** techniques observe program behavior during execution and can detect concrete vulnerabilities, though limited to actually executed paths, and usually come with a runtime overhead cost.

3. **Hybrid techniques** attempt to combine both static and dynamic methods, leveraging the strengths of each to achieve higher precision and better code coverage.

Compilers are a specific class of programs using program analysis and holding a key position in the software development lifecycle. In a nutshell, they translate high-level human-readable source code into low-level machine code. However, they usually not only translate the code, but also optimize it thanks to a broad set of static analysis techniques coming from compiler theory, as illustrated by the *Dragon Book* from A. V. Aho *et al.* [14].

GCC is one of the most widely used compilers, especially for C code, according to a 2023 survey on developers ecosystem from JetBrains [15], supporting numerous programming languages and architectures. Within GCC, its static analyzer provides a symbolic execution engine capable of exploring the different paths in C programs starting with GCC 10 [16]. Succinctly, the GSA is a static analysis framework abstracting GCC's internal data structures and leveraging a symbolic execution engine and data structures to implement a broad set of analyses, thus allowing to detect vulnerabilities at compile-time [17].

In this thesis, we focus on static analysis within the GCC compiler itself by extending its static analyzer to detect the aforementioned secret-related weaknesses. By dissecting its architecture and extending its capabilities, we aim to adapt the GSA to the specific problem of secret-related weaknesses. Our contributions demonstrate how compiler-integrated analysis can both detect missing zeroization in practice, namely CWE-226 and CWE-244, which can then be generalized to broader categories of sensitive information leakage, i.e., descendant of CWE-200. Thanks to our work, the GSA turns out to not only be a tool for general bug detection but also a promising platform for refining the state of software security.

The choice of the GSA is motivated by several aspects:

1. Developers tend to dislike tools that break their workflow, according to various surveys [18–20]. However, compilers are already part of the software development workflow and lifecycle, making them natural hosts for static analysis techniques willing to enhance software security at scale.
2. GCC's monolithic architecture allows it to be responsible for every code transformation, without relying on other programs' analysis. In contrast to the Clang/LLVM relationship, Clang relies primarily on code transformations provided by the LLVM compiler rather than within Clang itself.
3. The GSA is placed at an interesting place within GCC's compilation pipeline, i.e., at the late stage of the language and machine-independent passes. So far, it only supports analysis of C code, though there is ongoing work to extend it to C++ for the GCC 16 release. However, it remains a promising static analysis framework embedded directly inside the original compiler of the GNU/Linux Kernel.

1.1. Contribution of this Thesis

In this thesis, we investigate how compiler-integrated static analysis can be extended to address secret-related vulnerabilities. We approach this investigation through three research questions, each guiding one of our main contributions:

I. *How suitable is the GSA as a foundation for security-related analyses?*

II. *Can an analysis, focusing on an information leakage use-case, be implemented within the GSA?*

III. *How can a GSA-based analysis be generalized to detect broader information leakage weaknesses?*

The contributions of this thesis are divided into three complementary categories. First, we dissect the GSA to evaluate its architecture and internals, thereby assessing its suitability as a foundation for security-oriented analyses. Second, we extend the analyzer with GnuZero, a tool to detect missing zeroization of sensitive data in C code, focusing on CWE-226 and CWE-244. This contribution introduces new language attributes and a taint-propagation model, leverages the GSA's expressive reporting system, and is validated on both benchmarks and real-world software projects. Finally, we propose GnuSecret as a generalization of this approach, unifying the treatment of secret-related weaknesses under a new set of C attributes. By propagating secrets throughout the program's variables and detecting unsafe flows to different sinks, GnuSecret demonstrates its ability to capture broader vulnerabilities such as those falling under the scope of CWE-200, spotting both known CVEs and new ones in complex software. Together, these contributions illustrate the feasibility of embedding security-oriented analysis directly into a mainstream compiler.

1.1.1. Dissecting the GSA

The first contribution of this thesis acts as a foundational work for the following contributions, and is an in-depth dissection of the GCC Static Analyzer (GSA). This diving within its internals clarifies its symbolic execution

engine, memory model, reporting system, and overall architecture. Through this analysis, we establish the foundation on which new vulnerability detection mechanisms can be reliably built, while highlighting the strengths and limitations of the GSA. Overall, this contribution aims to answer question **I** and motivates its extension for information leakage use cases.

As part of this dissection, we describe the exploded supergraph that the GSA uses for interprocedural control-flow analysis, along with its region-based memory model, which is used to represent stack and heap layouts. We also examine the GSA's reporting system, showing how warnings are constructed and traced to source locations, and the interaction between the GNU attribute system and the GSA, identifying how attributes can be used to extend language semantics to drive analysis.

To assess the GSA effectiveness to work on real-world codebases, we evaluated the GSA on a subset of the Juliet Testsuite [21], specifically the subset targeting both CWE-214 (*Double Free*) and CWE-215 (*Use-After-Free*). We also reproduce the detection of CVE-2020-1967 within OpenSSL [22], initially detected by the GSA.

To resume this contribution, we document and detail the GSA's internals, creating a technical centralized documentation sourced by the plurality of documentation from the GCC's project. It is the foundational work of the following contributions.

1.1.2. GnuZero: A Static Tool to Detect Non-Zeroization

Zeroization, or scrubbing, is the practice of explicitly erasing sensitive information from memory once it is no longer needed. It is an information leakage weakness which is the use-case chosen to answer question **II**. The objective is to ensure that secrets do not persist in program memory beyond their intended lifetime, where attackers could later recover them. Naively, it is a simple problem: erase the sensitive data. Unfortunately, zeroization is hard to enforce in practice [23]. Even when developers call explicit scrubbing functions, compiler optimizations may silently remove them if. Indeed, aggressively optimizing compilers might detect write operations on memory objects that will not be read afterwards. This is known as the Dead-Store Elimination (DSE) problem. Furthermore, secrets propagate through compiler-inserted temporary variables or function parameters, making it painful for developers to track all memory objects that require zeroization.

Based on the conclusion from the dissection of the GSA, our second contribution is GnuZero, a compiler-based tool to detect missing zeroization of sensitive data. Targeting CWE-226 and CWE-244, GnuZero extends the GNU attribute system to enable developers to drive our analysis. We introduce two attributes: `scrub` to mark sensitive variables and `scrubber` to identify zeroization functions. On top of this, we define a taint-propagation model to automatically identify both stack and heap variables left uncleared at the end of their lifetime. Our evaluation shows that GnuZero achieves full coverage on a benchmark dedicated to CWEs, namely the Juliet Testsuite [21]. We also evaluate GnuZero on known CVEs and detect new information leakage vulnerabilities within real-world projects.

> **Reference:** This work is the outcome of a joint work with Mohamed Sabt and Pierre-Alain Fouque. It has been published in the proceedings of the *IEEE/IFIP International Conference on Dependable Systems and Networks 2025* [24], and received one of the three *Best Papers Awards* and the *Distinguished Paper Award*.

1.1.3. GnuSecret

Our first contribution confirmed that the GSA was a good fit for an analysis targeting both CWE-226 and CWE-244 directly within the GCC compiler. However, the spectrum of weaknesses related to sensitive information disclosure is broader, as illustrated by the numerous children of CWE-200 [13]. Several tools from the research community exist that target different information leakage vulnerabilities, despite their similarities. Worse, each of them is coming with their own annotation system, reporting system, and static analysis techniques [25–29], increasing the confusion among the concerned developers [30].

Our final contribution, called GnuSecret, is a framework that generalizes GnuZero's approach to broader information disclosure vulnerabilities and aim at answering to research question **III**. By introducing several secret-related attributes, it unifies the semantics for secrecy tracking across variables, functions, and C types. Combined with the symbolic execution engine of the analyzer, GnuSecret propagates secrets based on the program's data-flow and detects unsafe flows to leakage sinks. To evaluate GnuSecret, we analyzed open-source projects known to be vulnerable to assess GnuSecret's efficiency: GnuTLS [31], OpenSC [32], and libJWT [33]. Finally, GnuSecret detected an unknown vulnerability in a cryptographic module of the GNU/linux kernel. Both the analysis

of known vulnerable code and the detection of a new vulnerability illustrate the effectiveness of GnuSecret's approach in practice.

> **Reference:** This work is the result of a joint work with Mohamed Sabt, Abdoulaye Katchala Mele, and Pierre-Alain Fouque. It is the subject of an ongoing submission in an academic conference.

1.2. Artifacts

In an effort to promote open science, all our contributions feature publicly available artifacts on Zenodo and Git repositories on the INRIA GitLab instance. A super archive has been published on Zenodo [34], containing every artifact of this thesis, including the artifacts of the GSA evaluation, GnuZero, and GnuSecret.

Custom GCC: based on a specific commit [35], the source code is available at https://gitlab.inria.fr/pphilipp/gcc/-/tree/analyzer-tracking-region?ref_type=heads. Our modifications have been presented in the FOSDEM technical conference in 2024 [36] to the GCC developers community [37].

GSA Evaluation: introduced in Section 1.1.1. An archive is available on Zenodo [38], containing:

1. sources of the Juliet test suite for CWE-214 and CWE-215, alongside SARIF files from the analysis and a Python script to run the tests,
2. and a cloned repository of OpenSSL version 1.1.1f, along with a script to reproduce the analysis that led to the detection of CVE-2020-1967 [22].

GnuZero: introduced in Section 1.1.2. GnuZero's source code is available at https://gitlab.inria.fr/pphilipp/plugin-scrub. Alongside the source code, an archive specific to GnuZero's evaluation is available on Zenodo [39] containing:

1. the modified version of GCC at the time of publication,
2. utilities to build from source a Debian package containing the compiled modified version of GCC (though an already built package is present),
3. the adapted sources of the Juliet test suite for CWE-226 and CWE-244, alongside SARIF files from the analysis and a Python script to run the tests,
4. and the source code of GnuZero and the different tests.

GnuSecret: introduced in Section 1.1.2. GnuSecret's source code is available at https://gitlab.inria.fr/pphilipp/gcc-plugin-ct. Alongside the source code, an archive specific to GnuSecret's evaluation is available on Zenodo [40] containing:

1. a modified version of the GCC compiler,
2. utilities to build from source a Debian package containing the compiled modified version of GCC (though an already built package is present),
3. the source code of GnuSecret,
4. and the adapted source code of the analyzed libraries, accompanied by SARIF output files and a Python script for executing the different tests.

1.3. Outline

This thesis is divided into three parts, excluding this introduction. Part I introduces some background on central notions to our contributions:

- Chapter 2 introduces GCC's architecture and its intermediate representations.
- Chapter 3 presents foundational concepts in program analysis, with a strong focus on static analysis.

Next, the central part of this manuscript details our contributions in Part II:

- Chapter 4 dissect the GSA's internals, including key data structures, its symbolic execution engine and reporting system.
- Chapter 5 presents GnuZero's leveraging the GSA internals to detect missing zeroization even in optimized builds.
- Chapter 6 presents GnuSecret, a generalized framework based on the GSA to detect diverse information leakage sinks.

And finally, Part III summarizes the thesis contributions, and outlines directions for future research and engineering efforts.

How to read. At the onset of each chapter, readers are provided with a concise overview underscoring the main themes or contributions. Moreover, at the beginning of technical sections, *Takeaway boxes* are provided to summarize the section's content, as shown in the example below.

Takeaway: This is a takeaway box.

Some reminders briefly revisit essential concepts explained are provided in *Reminder boxes*, as illustrated below.

Reminder: This is a reminder box.

Part I

Preliminaries

Chapter 2

GCC: The GNU Compiler Collection

Introduced in 1987, the GCC is a compilers suite widely known for being the historical compiler for projects such as the GNU/Linux kernel. Essentially, a compiler is expected to take a set of files in a given source language as input and output a binary version of the program tailored to a specific architecture. The compiler's magic occurs within the compiler itself and is transparent to users seeking a reliable working compiler without requiring any knowledge of compilation's arcane.

GCC is a rather *monolithic* compiler compared to other famous compiler suites, such as the LLVM-based family (including Clang and Rust). Monolithic here refers to a single, integrated system that serves as a black box to average end users. It is the same with LLVM-based compilers, though LLVM frontends usually rely on optimization passes from the LLVM compiler itself, apart from potential language-dependent optimizations. In contrast, GCC is a framework that holds the entire compilation logic, from source validation to binary emission (c.f., Figure 1).

As a 40-year-old project, GCC was first written in C, and an effort was made to modernize its inner architecture; part of its internals were rewritten in C++. The fact that it was first written in C remains heavily present at the core level of the compiler. However, before diving into its software architecture, it is essential to discuss its various Intermediate Representations (IRs), or Intermediate Languages (ILs) as they are referred to in GCC's internals manual [41]. Quoting this manual, there "*are many places in which this document is incomplet and incorrekt* [sic]. *It is, as of yet, only preliminary documentation*". Hence, understanding GCC internals and how programs are represented, it is almost mandatory to dive into its source code.

This chapter presents the different components of GCC based on the work carried out during this thesis and is based upon a specific GCC commit [35].
Firstly, Section 2.1 is resuming the different evolutions GCC went through over time regarding its IRs. Section 2.2 dives into the concepts, data structures, and IRs involved in GCC. Section 2.3 details the GNU attribute system, Section 2.4 discuss GCC's warning engine limitations and Section 2.5 describes GCC's plugin architecture.

Contents

2.1. A Brief History of GCC 29
2.2. GCC Internal Representations 29
2.2.1. GENERIC 29
2.2.2. GIMPLE 32
2.2.3. RTL 35
2.3. GNU Attribute System 36
2.3.1. Examples of Attributes 37
2.4. GCC's Warning Engine Limitations 38
2.5. Plugins 38

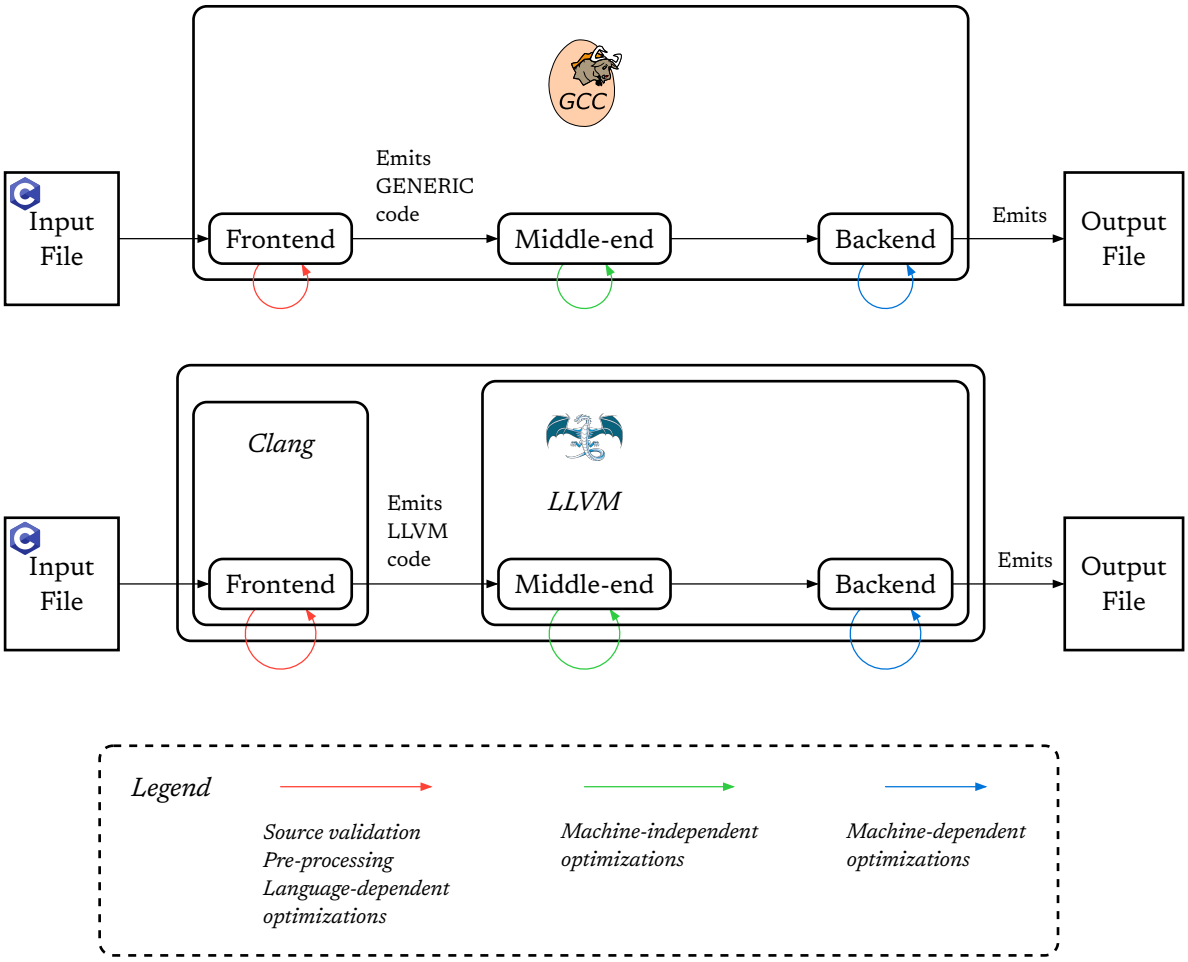


Figure 1: Compilation of a C file using GCC and Clang, from a black-box approach.
Both compiler architectures are partitioned into three major components: a frontend for source validation, a middle-end for machine-independent optimizations, and a backend for machine-dependent optimizations and binary emission. The key difference is that GCC is a standalone monolithic compiler, whereas Clang serves as a C frontend for the LLVM compiler.

2.1. A Brief History of GCC

Since its initial release in 1987, GCC's internals have undergone a lot of changes. In the same manner, a high amount of materials were produced by the developers, including documentation, email exchanges, and technical articles submitted to the discontinued GCC Summit (now replaced by the GNU Tools Cauldron). The plurality of sources makes it challenging to get a handle on GCC because it requires going through a lot of documentation, technical articles, comments in the source code, directly in the project's git commit messages, or even in the mailing list history. All this motivated the present section to gather in a relatively short manner the IRs evolution within GCC.

In Section 2.2, we present the three main IRs within GCC, but initially, GIMPLE and GENERIC did not exist. Instead, frontends were generating an Abstract Syntax Tree (AST), relatively close to GENERIC, but which was not an IR by itself. At that time, GCC was directly translating each statement of this AST into Register Transfer Language (RTL), and all optimizations and analysis were performed on this low-level IR.
Alas, RTL was not always fitting the needs of some analyses, making them even more costly (e.g., function inlining) [42]. In parallel, as RTL is a complex IR embedding information on the targeted architecture, the complexity for new maintainers to enter the source code was high.

During the years 2003-2006, motivated by these factors, work was carried out to introduce GENERIC and GIMPLE [42], including its Static Single-Assignment (SSA) form [43–45], becoming the first IR under SSA form in GCC. Formerly, GIMPLE shared the same architecture as GENERIC: a single union with several structs and a base structure contained in all the other members of the union, as presented in Section 2.2.1. But in 2013, work was accomplished to transform this union into a C++ structure hierarchy[1].

In 2020, mechanisms to transform RTL under an SSA form[2] is added to GCC. It remains limited, since the RTL SSA form is far more restrictive than GIMPLE SSA. The different RTL passes are responsible for the construction of the RTL SSA and release it before the end of their execution, whereas GIMPLE SSA "survives" from a GIMPLE SSA pass to another.

2.2. GCC Internal Representations

> **Takeaway:** GCC relies on three main IRs:
> 1. GENERIC, an AST closed to the source language and with instructions similar to C ones. It is the IR generated by the different frontends.
> 2. GIMPLE, an IR language and machine-independent, limited to one side-effect per instruction. This IR is at the heart of the middle-end, as many machine-independent optimizations are performed on it.
> 3. RTL is the final language-independent IR, the core of GCC's backend. It is associated with a virtual machine that has an infinite number of registers and is aware of the targeted architecture, leveraging machine-dependent optimizations.

From its internals manual, GCC relies on "three main intermediate languages to represent the program during compilation: GENERIC, GIMPLE, and RTL.
This section will dive into the different IRs involved in GCC's compilation pipeline: GENERIC in Section 2.2.1 and GIMPLE in Section 2.2.2, which consists of the main IRs of concern for this thesis, and then, Section 2.2.3 briefly introduces RTL.

2.2.1. GENERIC

> **Takeaway:** This IR is internally represented by a C-tagged union `tree`, at the heart of GCC, used to describe everything in the AST, including, but not limited to, variables, functions, attributes, types, or even expressions. In a nutshell, it does have an internal type system, not reflected by the C type system, since it uses a discriminant `enum` to distinguish between the types of trees. Most access to inner fields is performed through the use of C macros, which can potentially lead to runtime crashes if not used with the correct tree type.

This IR is language-independent and mainly used as "an interface between the parser and optimizer" and can represent all "programs written in all the languages supported by GCC" [41]. It relies on a single C union to represent every token, statement, and expression in a program source.

1. Commit ID: daa6e488c2f9611ac88611aac6f49e4090eb4902
2. Commit ID: 73b7582775254b764fd92ddb252a33dc15872c69

2.2.1.1. Cutting Down The Tree

One data structure is at the heart of this representation: the `tree` type, a pointer to the `tree_node` union (c.f., Listing 1). We will now use the same convention as the GCC internals manual and will refer to trees in ordinary font, except when talking about the actual C type `tree`.

```c
union tree_node;
typedef union tree_node *tree;
```

Listing 1: Definition of the `tree` type.

Essentially, a tree is a node within the IR AST and can represent anything, including a constant, a variable, a function, a type, or even an expression. Figure 2 depicts a simple C expression and its tree representation. A node is a C union with members consisting of 39 structures of varying sizes, which can lead to runtime errors, as most of the accessors to a tree member are implemented as C macros.

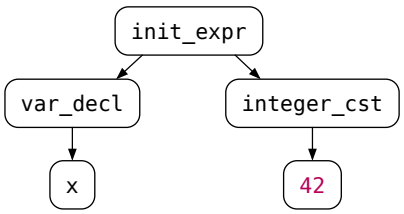
The `tree_node` union is part of GCC's legacy. Its inner representation is written in C, and the idea behind this union is to mimic a class hierarchy within its members using a tagged union approach. All of its members are structures based on a single structure: the `tree_base` structure. This base holds several pieces of information that make sense only in some instances (e.g., a flag to identify a constant or another for a volatile variable). Thanks to the first element of the `tree_base` structure, a 16-bit-wide enum `tree_code`, it is possible to discriminate between the different kinds of trees.
As previously mentioned, most of the accessors are C macros. The `TREE_CODE` macro is the accessor to the `tree_code` of a given tree, which, when successfully compared to a known `tree_code`, allows the safe usage of the accessors corresponding to that kind of tree. Listing 2 illustrates how this macro is defined in GCC's source code, alongside some usage examples to discriminate a given tree. To reformulate, the tree system holds its own "*type-system[...], but it is not reflected in the C type-system*" [41, §10.2].

The complexity of trees is illustrated in Figure 3: a single node can represent a variable, a type declaration, or even specific optimizations of a function. Their name is explicit and built in the form of *tree_something_decl*, meaning it is a tree representing something declared. For example, *tree_function_decl* is the tree of a declared function, and *tree_var_decl* represents a declared variable (either global or local). The same applies to *tree_parm_decl*, *tree_return_decl*, and *tree_type_decl*, which represent a function parameter, a returned variable, and a declared type (e.g., through `typedef`). Of particular interest, *tree_field_decl* is a tree representing the field associated with a structure, and *tree_ssa_name* is a variable under the SSA form. Those are essentially the base trees used by GCC, on top of which specific operations can be used by combining `TREE_CODE` and base tree(s). For example, an array access is represented by a tree with a `TREE_CODE` `ARRAY_REF` and two subtrees:

1. the `tree_var_decl` representing the array indexed

```c
int x = 42;
```

Subfigure 2.1



Subfigure 2.2

Figure 2: A simple C expression in Subfigure 2.1, broken down in a tree representation using GENERIC `tree` in Subfigure 2.2.

```c
/* Definition of the TREE_CODE macro (from file gcc/tree.h). */
#define TREE_CODE(NODE) ((enum tree_code) (NODE)->base.code)
/* Function to determine if a tree represents a constant integer (e.g., "42"). */
bool is_a_constant (tree t) {
  return TREE_CODE (t) == INTEGER_CST;
}
/* Function to determine if a tree represents a sum expression. */
bool is_a_sum_expr (tree t) {
  return TREE_CODE (t) == PLUS_EXPR;
}
```

Listing 2: Definition of the `TREE_CODE` macro and usage examples.

2. and either:
   - an `integer_cst` representing the index for constant access,
   - or the tree corresponding to the variable used as index for dynamic access.

The idea behind the usage of trees is similar to the logic of memory management in C; the developer must be aware of what they are doing, otherwise risking Undefined Behavior (UB). When using trees, this can be seen as a type confusion: accessing a non-existent field of a specific structure's instance might crash instantly or at a later point during execution. Thankfully, GCC developers considered that risky behavior, and it is possible to compile GCC in a way that every macro accessor checks whether its argument is one of the expected kinds; otherwise, an internal compiler error is thrown, explaining the issue, mainly to detect misuse of those accessors in early development stages.

2.2.1.2. Purpose and Compilation Pipeline Relationship

GENERIC is an early IR, designed to represent an entire function in a language-independent manner, and generated by the different source language frontends handled by GCC (e.g., C, C++, or Cobol). No language-independent optimizations are performed on it. Frontends are responsible for generating a valid GENERIC code, and to do this, they usually perform syntactic and grammatical validation of the source code. Before the final stages of GENERIC, frontends can perform any necessary tasks, such as pre-processing (e.g., C/C++ frontends) and potential language-dependent optimizations (e.g., the under-development Rust frontend). Still, GENERIC is

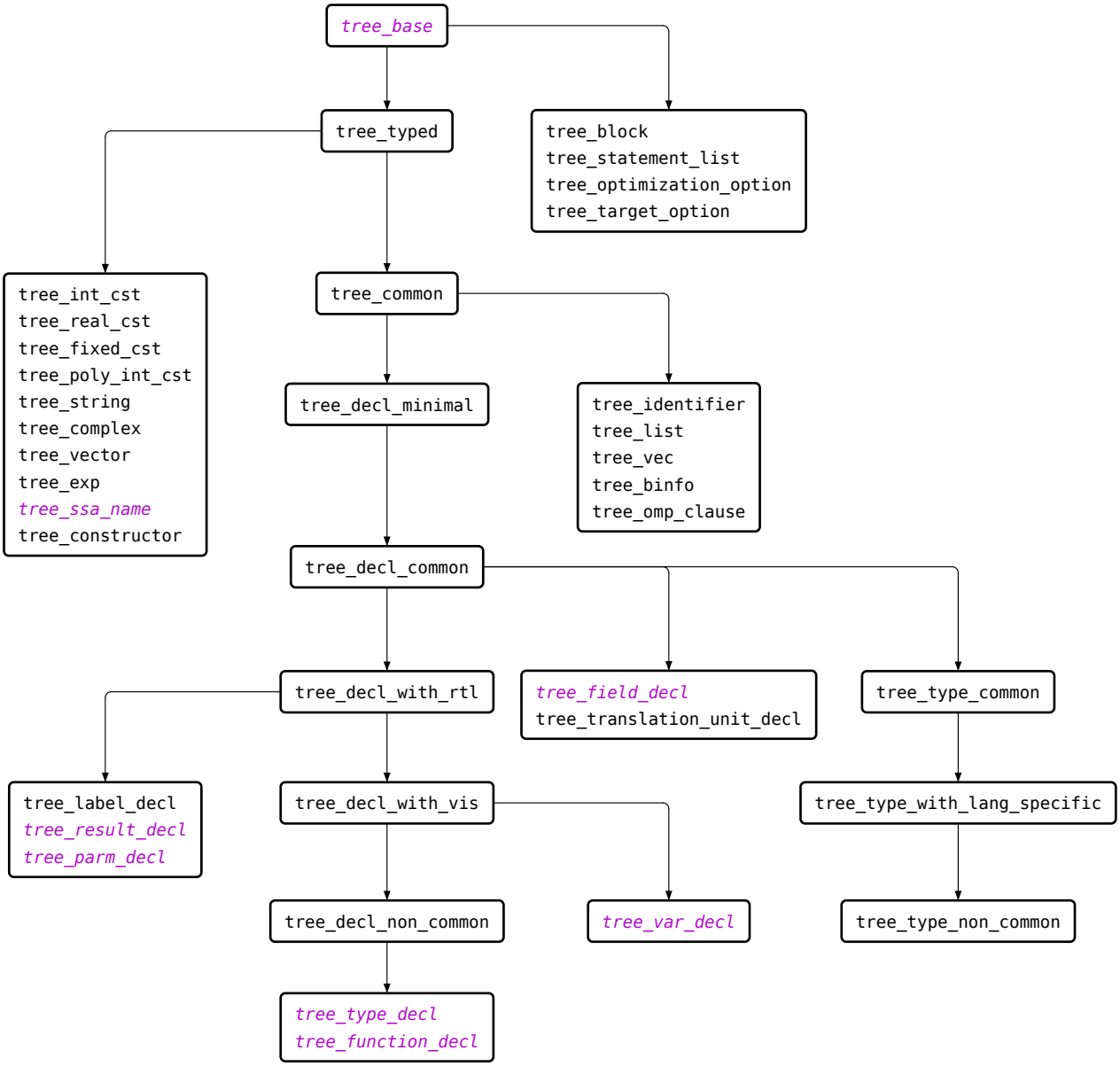


Figure 3: The structure hierarchy behind the `tree_node` union, starting from `tree_base`. The members of specific concern have been *emphasized like this* for clarity.

considered part of GCC's middle-end and not its frontend. It is emitted by the frontends and passed as input to the middle-end, same as LLVM IR is generated by Clang and passed as input to the LLVM compiler.

2.2.2. GIMPLE

> **Takeaway:** GIMPLE is a language and machine-independent IR used by GCC's middle-end to perform various optimizations such as Dead-Store Elimination (DSE), Constant Propagation, or Loop Unwinding. During the interprocedural passes, it is transformed under the Static Single-Assignment (SSA) form to ease the workload of optimization passes.

As mentioned earlier, GENERIC and its data structures are part of GCC legacy: originally written in C, there is no real way to implement a class hierarchy besides using unions and enums to discriminate between instances. It is harsh to use and master for newcomers, primarily due to the complexity of the data structure, which motivated the introduction of a more C++-oriented IR: GIMPLE.

GIMPLE is heavily influenced by SIMPLE, an IR designed for the McCAT compiler [46], with different design choices. For example, SIMPLE has no support for `goto` statements, whereas GIMPLE does.

Genuinely, GIMPLE is not a single IR, but a set of several ones. This section explains the different stages that the IR undergoes during compilation, with a focus on the major ones: High GIMPLE, Low GIMPLE, and GIMPLE SSA. Section 2.2.2.1 introduces the lowering from GENERIC to GIMPLE, highlighting the differences between High and Low GIMPLE. Section 2.2.2.2 then dives into GIMPLE under the Static Single-Assignment (SSA) form.

2.2.2.1. Gimplification

The process by which GENERIC is lowered to GIMPLE is called *gimplification* and corresponds to the compiler pass referred to as the *gimplifier* within the GCC source code. As previously mentioned, GIMPLE is separated into two different IRs: High and Low. The *gimplifier* is lowering GENERIC to High GIMPLE, and although it is referred to as a pass, it is not an actual pass. Indeed, it is more a set of functions exposed to other components of the compiler and passes. Gimplification occurs during the first stages of constructing the symbol table for compilation units, specifically during the initial call graph computation, which takes place immediately after source validation and pre-processing for C code. Interestingly, variable declarations and their initializations are separated in GIMPLE.

> **Reminder:** A call graph is a Control-Flow Graph (CFG) representing the relationship between functions of a program. Each node in a call graph represents a function, and an edge between two nodes f and g indicates that f calls g.

Essentially, GIMPLE instructions, more restrictive than GENERIC ones, have no more than three operands per statement, with a few exceptions (e.g., function calls), and complex loops (i.e., `for` and `while`) are broken down into `if` and `goto` instructions. A valid GIMPLE statement should not contain more than one side-effect instruction. If needed, the gimplification process might introduce anonymous variables, i.e., variables not present at the source level, but required to compute a complex expression (c.f., Figure 4).

Following the same logic, Low GIMPLE is more restrictive than High GIMPLE, for example, Exception Handler (EH) are present in High GIMPLE but not in Low. Another distinction is that High GIMPLE still maintains a notion of local lexical scope within functions. In contrast, Low GIMPLE has a single lexical scope at the function level, which is closer to the semantics of machine code. Table 1 is a non-exhaustive table of the instruction set for both High and Low GIMPLE.

High GIMPLE does not exist very long during the compilation process. The last pass, named `pass_lower_cf` and lowering from High GIMPLE to Low, is executed before any optimization passes. Figure 5 illustrates an example

```c
1  int x = *ptr++;
```
C

Subfigure 4.1

```
1  int _1 = *ptr;
2  int x = _1 + 1;
```
High GIMPLE

Subfigure 4.2

Figure 4: A single C statement with two side-effects (pointer dereference and sum) in Subfigure 4.1 and its GIMPLE version in Subfigure 4.2, composed of two statements introducing the anonymous variable _1.

of the gimplification process from source level to High and Low GIMPLE. From now on, GIMPLE will specifically refer to Low GIMPLE.

Instruction	Sub-statements	Description	High GIMPLE	Low GIMPLE
GIMPLE_ASSIGN	No	Assign statement	✓	✓
GIMPLE_BIND	Yes	Represent a lexical scope	✓	⊘
GIMPLE_CALL	No	Function call statement	✓	✓
GIMPLE_TRY_CATCH	Yes	Represent EH	✓	⊘
GIMPLE_COND	No	Condition statement	✓	✓
GIMPLE_GOTO	No	Jump statement	✓	✓
GIMPLE_LABEL	No	Represent a label	✓	✓
GIMPLE_PHI	No	Represent a Φ function	⊘	✓
GIMPLE_RETURN	No	Return statement	✓	✓
GIMPLE_SWITCH	No	Switch statement	✓	✓

Table 1: High GIMPLE and Low GIMPLE instruction set. Note: only the instructions related to C are present; C++ and OpenMP (OMP) instructions have been removed for clarity.

High GIMPLE

```
1  try
2  {
3    a = 7;
4    b = 5;
5    c = 42;
6    ptr = &a;
7    a.0_1 = a;
8    _2 = b * a.0_1;
9    _3 = *ptr;
10   _4 = _3 / c;
11   x = _2 + _4;
12 }
13 finally
14 {
15   a = {CLOBBER(eol)};
16 }
```

Subfigure 5.1

C

```
1 int a = 7;
2 int b = 5;
3 int c = 42;
4 int *ptr = &a;
5 int x = a * b + *ptr / c;
```

Subfigure 5.2

Low GIMPLE

```
1  a = 7;
2  b = 5;
3  c = 42;
4  ptr = &a;
5  a.0_1 = a;
6  _2 = b * a.0_1;
7  _3 = *ptr;
8  _4 = _3 / c;
9  x = _2 + _4;
10 a = {CLOBBER(eol)};
```

Subfigure 5.3

Figure 5: Listings representing the same C program Subfigure 5.2, both in High GIMPLE Subfigure 5.1 and Low GIMPLE Subfigure 5.3. An example of EH is illustrated in the High GIMPLE representation, using a `try/finally` statement, and is due to variable a's address being taken.

2.2.2.2. GIMPLE SSA

Static Single-Assignment (SSA) form [47,48] represents a program while easing static analysis by modifying a program, such that each variable is assigned only once. LLVM IR is by design under SSA form [49], whereas GIMPLE is not. SSA implementation in GCC comes from the work of R. Cytron *et al.* [50], on top of which LLVM is also based.
The pass applying the SSA form to GIMPLE is an inter-procedural pass named `ipa-build-ssa-passes`. This pass does not modify the code, but instead relies on a set of sub-passes responsible for transforming functions into an SSA form. The main subpass of interest here is the `tree-ssa` pass, responsible for the actual SSA form transformation of the GIMPLE representation of functions.

```
1  a = 7;                                   GIMPLE SSA
2  b_7 = 5;
3  c_8 = 42;
4  ptr_9 = &a;
5  a.0_1 = a;
6  _2 = b_7 * a.0_1;
7  _3 = *ptr_9;
8  _4 = _3 / c_8;
9  x_10 = _2 + _4;
10 a ={v} {CLOBBER(eol)};
11 return;
```

Listing 3: SSA form of program from Figure 5.

To respect SSA form, transformations need to be applied to GIMPLE variables that have multiple assignment sites: for each of them, the compiler creates a new version of the given variable. Still, variables assigned only once are also transformed in the same way. In GIMPLE SSA, the different versions of a variable are distinguished by subscripting the variable name: a variable x with two assignment sites will have two versions (e.g., x_1 and x_2, the latter being more recent). The most recent SSA version of a given variable "invalidates" the usage of the older ones. Thus, when a given variable is used in the right-hand side (RHS) of an assignment, the compiler will use its more recent version.

Alas, programs are not always linear; the control flow might make it impossible for the compiler to choose which version of a given variable to use (e.g., loops). In those cases, the compiler will insert a PHI (Φ) function, creating a new version of the variable, with all incoming versions as parameters.

Reminder:

A Φ-function is a no-op function designed for SSA [50] to merge branches joining at a later node of a CFG assigning to the same variable. Below is a C program with an assignment to x depending on a condition.
On the right is the equivalent program in GIMPLE SSA. Note that the source node's ID of a variable's version is present in the Φ node on line 17 (i.e., basic block 3 for x_6 and 4 for x_5, respectively assigned on line 10 and 14).

```
1 extern int some_cond;                     C
2 int x = 42;
3 if (some_cond)
4   x--;
5 else
6   x++;
7 int y = x;
```

```
1  <bb 2> :                                 GIMPLE SSA
2  x_3 = 42;
3  some_cond.0_1 = some_cond;
4  if (some_cond.0_1 != 0)
5    goto <bb 3>; [INV]
6  else
7    goto <bb 4>; [INV]
8
9  <bb 3> :
10 x_6 = x_3 + -1;
11 goto <bb 5>; [INV]
12
13 <bb 4> :
14 x_5 = x_3 + 1;
15
16 <bb 5> :
17 # x_2 = PHI <x_6(3), x_5(4)>
18 y_7 = x_2;
19 return;
```

Nevertheless, some variables cannot be put in SSA form if they are either: 1) global, 2) volatile, 3) aggregate (i.e., a member of a struct or an element of an array), or 4) need to live in memory (i.e., their address is taken). Those limit cases are because the compiler must be aware of all its assignment sites to generate the SSA version of a variable, as discussed with GCC's developers [51].

GCC choose this design for global and volatile variables, as those are potentially shared memory. Regarding aggregate variables, arrays are either indexed with a constant or a local variable, probably under SSA form. In both

```c
int x = 42;
int *ptr = &x;
x++;
```

C

Subfigure 6.1

```
x = 42;
ptr_5 = &x;
x.0_1 = x;
_2 = x.0_1 + 1;
x = _2;
x ={v} {CLOBBER(eol)};
```

GIMPLE SSA

Subfigure 6.2

Figure 6: A C program in Subfigure 6.1 and its GIMPLE SSA form in Subfigure 6.2. Variable x remains under non-SSA form due to its address being taken.

cases, GCC does not consider elements of an array as a classic variable, and no new version of it is generated upon a new assignment. This behavior is the same for structures or unions, as they are considered as arrays.

Regarding the last case, it is a design choice made by the compiler. Truthfully, for variables with their address taken that are either global or volatile, the issue is the same as earlier. Whereas for local variables, one could argue that it should be possible to transform them into an SSA form. Nonetheless, a problem arises when considering the variable holding the address of the new SSA variable: to which version does it point? Each new version of this variable would invalidate previous pointers assigned with earlier versions addresses. This behavior is illustrated both in Listing 3 with variable a and in Figure 6 with variable x.

Comparison to LLVM. GIMPLE SSA and LLVM IR differ on their semantics for assignment statements (including initialization) and variable declarations. Regarding GIMPLE, there is no representation of a declaration statement; instead, global variables are enclosed in the scope of the translation unit, and local variables in their enclosing function's scope. An assignment statement is represented with the *simple assignment* operator, defined in the C standard [52, §6.5.17.2]. During its transformation under the SSA form, GIMPLE's assignment statement follows the previously defined rules. On the other hand, LLVM is designed to be in SSA form. It defines instructions resulting in the allocation of virtual registers (e.g., the `alloca` or the `load` instructions), whereas the assign statement relies on the `store` instruction [53]. Although all virtual registers are indeed assigned once through allocation instructions, their content can still change due to the semantics of the `store` instruction. The `store` instruction does not return a new virtual register; instead, it takes as an argument the register to be overwritten, alongside its new content. Figure 7 illustrates the differences between the two IRs. The same virtual register is used both times as an argument to the `store` instructions in LLVM IR (c.f., Subfigure 6.2), whereas two distinct SSA versions of the same variable are assigned in GIMPLE SSA (c.f., Subfigure 6.3).

As previously mentioned, the SSA form leverages some analysis by simplifying certain parts of the work (e.g., constant propagation). Upon completion of the analysis passes relying on the SSA form, the program is transformed back into a non-SSA form, before being lowered to RTL, following an algorithm proposed by R. Morgan [54, Section 7.3].

2.2.3. RTL

Takeaway: Within GCC, RTL is a Lisp-inspired S-expression language and is the final IR on which architecture-dependent optimizations are performed before actual machine code emission.

The idea behind Register Transfer Language (RTL) essentially comes from the work of J. W. Davidson *et al.* [55]. They describe a machine as "a grammar for syntax-directed translation between assembly language and register transfers". Essentially, RTL is associated with a virtual machine composed of an architecture-independent assembly language and an infinite set of registers, and is a common IR within compilers [56,57].

```c
int x = 42;
x = 5;
```

C

Subfigure 7.1

```
%x = alloca i32, align 4
store i32 42, ptr %x, align 4
store i32 5, ptr %x, align 4
```

LLVM

Subfigure 7.2

```
x_1 = 42;
x_2 = 5;
```

GIMPLE SSA

Subfigure 7.3

Figure 7: A simple C program presentend in Subfigure 7.1 and its corresponding program in GIMPLE SSA and in LLVM IR, respectively in Subfigure 7.3 and Subfigure 7.2.

```
(set                    RTL
  (reg:SI 140)
    (plus:SI
      (reg:SI 138)
      (reg:SI 139)
    )
  )
)
```

Listing 4: RTL instruction adding two registers (138 and 139) and storing the result in register 140.

RTL in GCC is the low-level IR used in the backend to describe a program, and each operation is in an assembly-like language.
In practice, RTL is a Lisp-inspired S-expression language: each instruction is a `(set …)` expression with nested sub-expressions for operands (c.f., Listing 4). It encodes target-specific details (e.g., register classes and calling conventions) directly in the IR.

RTL IR in GCC does not escape the rule; it is an abstract assembly for a virtual machine with infinite registers, before registers are allocated to real hardware [41].

It is possible for RTL passes to transform RTL into an SSA form if needed, though there is no guarantee to be consistent from one pass to another, as optimization passes might change the CFG. Instead, if a pass needs RTL under SSA form, it must request to build it at the beginning of its execution and release it at the end [41].

Finally, after all RTL optimization passes have been executed, RTL is translated into the targeted assembly language by following the target Machine Description (MD), a file describing how to translate RTL instructions into the target instructions.

2.3. GNU Attribute System

Takeaway: The GNU attribute system extends the C standard, allowing developers to enrich the code with information about their intent without introducing new C keywords or breaking the program's semantics. In a nutshell, attributes can be applied to every C construct, from types to statements, and guide the compiler in optimization, while also helping to narrow down diagnostic emissions.

GCC defines an expressive attribute system on top of the C standard since GCC 2.9.3 [58]. Attributes allow developers to annotate some C constructs, such as variables, functions, or types, with extra information to help understand the code intent. Thus, they allow developers to introduce specific, implementation-defined behaviors to the language without having to introduce new syntactic changes, like adding a new keyword or augmenting the grammar accordingly [59]. Generally speaking, the compiler can use attributes to insert new code (e.g., the GNU attribute `strub`) or the static linker can modify the layout of the generated binary (e.g., the GNU attribute `ifunc`). However, in this manuscript, we will focus solely on attributes that are intended to be used in ways that have minimal semantic impact; i.e., the runtime functional behavior of an annotated code shall not change when the attributes are ignored, albeit slower or with less code verification. One could think of such attributes as comments for compilers, as they would inspect them at compile time, either to emit new (security-related) warnings or to make the existing ones more accurate by reducing their false positive rate. C (and C++) attributes are compiler-defined, unlike those in Java or C# that are user-defined. Therefore, unless extending the compiler, programmers cannot define their own attributes for use at runtime, because C does not retain attribute information in its compiled form.

The use of attributes is illustrated with `noreturn` (c.f. Listing 5), informing the compiler that a function will never return via normal execution flow. This attribute leverages the compiler to issue warnings when a function annotated with `noreturn` has paths that return normally. Additionally, the compiler could optimize out any unreachable code resulting from the call to this function or issue a warning to the developer.

```c
[[noreturn]] void my_function(void) {    C
  abort();
}
```

Listing 5: Usage of the `noreturn` attribute, borrowing the C23 syntax.

Either way has no semantic effect on the code. The deprecation of the keyword `_Noreturn` by the attribute `noreturn` in C23 highlights the new direction of the C language in which attributes are being promoted to be contextual keywords.

Being growingly used in major projects, like the GNU/Linux kernel [60], attributes are no longer defined only by GCC. Indeed, the lead maintainer of the Clang compiler, A. Ballman [61], has successfully strived to introduce attributes to the C23 standard [52].
Since their standardization, two sets of attributes are to be distinguished:

1. standard-defined with only seven attributes (e.g., `deprecated`);
2. vendor-defined, with GNU being the leading provider supporting more than 50 attributes within the GCC compiler.

Some projects, like the GNU/Linux kernel, define their attributes by extending the GCC system through plugins to, for instance, drive out additional compiler messages that might hint at some security issues during compilation.

2.3.1. Examples of Attributes

This section details the use and semantics of three additional attributes defined by C23, the GNU extension, and the GNU/Linux kernel.

2.3.1.1. C23 Attributes: `fallthrough` and `deprecated`

An example of a standardized attribute is `fallthrough` [52], which aims to mark a specific case within an enclosing switch statement as being reachable from an earlier case statement, allowing compilers to avoid emitting warnings for such events. This attribute is an example of an attribute designed to reduce false positive emissions and is illustrated in Subfigure 7.1.

On the other hand, the standardized attribute `deprecated` "*can be used to mark names and entities whose use is still allowed, but is discouraged*" [52, §6.7.13.5], allowing compilers to emit a warning upon usage of such objects or functions. Thus, this attribute aims at a new analysis type, uncorrelated to the dialect but tied to the software architecture itself, allowing developers to prepare for significant changes within their codebase.

2.3.1.2. The GNU Attribute `nonnull`

The relatively common `nonnull` attribute is a function attribute expecting as arguments the index(es) of the parameter(s) that should not be null.

At compile-time, GCC is then able to detect potential misuse of those functions if provided a (potentially) null pointer as one of their arguments (c.f. Listing 6).

```c
void [[nonnull(1)]] foo(void *a);
```

Listing 6: `nonnull` attribute usage example.

It is an example of an attribute that leverages new analysis from the compiler, allowing developers to express the assumptions of a given function more effectively.

2.3.1.3. The GNU/Linux Kernel Attribute `user`

Developers can enhance the GCC attribute system by adding their own attributes through its plugin API, as explained earlier. The kernel has several attributes defined within the project through "designed to provide a place to experiment with potential compiler features that are neither in GCC nor Clang upstream" [62] and are ported from grsecurity/PaX [63,64].

```c
switch (some_enum) {
  case A:
    // ...
[[fallthrough]]
  case B:
    // ...
}
```

Subfigure 8.1

```c
struct [[deprecated]] foo {
  int a;
  int *ptr;
}
[[deprecated]] bar (void) {
  // ...
}
```

Subfigure 8.2

Figure 8: Standardized attributes `fallthrough` Subfigure 8.1 and `deprecated` Subfigure 8.2 examples. Note that `fallthrough` is an expression attribute, whereas `deprecated` is a type or function attribute.

For example, the `user` attribute aims to implicitly initialize local variables "that could otherwise leak kernel stack to userland if they aren't properly initialized"[65]. The project implements a GCC plugin (c.f. Section 2.5) that provides a handler for this user-defined attribute, which will then add code initializing any variable, or part of them (e.g., `struct`) if needed. This attribute is an example of an attribute responsible for code generation.

2.4. GCC's Warning Engine Limitations

GCC's developer community has been working on enhancing issue detection at compile-time for about 10 years. However, as GCC is designed as an optimization-oriented compiler, it does not have a specific warning engine, but rather several warning passes that intervene at different moments in the pipeline, thus operating on different IRs (c.f., Section 2.2). This design implies limitations to implement warning emission passes within its compilation pipeline, as presented by M. Sebor [66], a GCC maintainer. In a nutshell, the reasons are plural:

1. Warning passes mainly are interprocedural passes, thus not interprocedural-aware.
2. Some optimization passes might insert or remove information or code in the program's representation, which can result in false-positive or false-negative warning emission.

In his blog post, M. Sebor illustrates and highlights various reasons for the limitations of the GCC's warning engine, with different outcomes. Here is a non-exhaustive list focusing on false-positive cases, i.e., no warning is emitted even though the C code presents UBs:

- **Aggressive Early Optimization**: to maximize code efficiency, GCC transforms aggressively the program's representation, potentially early in the compilation pipeline. Hence, information can be deleted from the IR, removing the compiler's capacity to detect the issue.
  M. Sebor [66] illustrates this behavior with an example regarding an optimization of a `strlen` usage with a known constant string to which is added an offset with an invalid value without any warning emission, as illustrated in Figure 9. The C code is depicted in Subfigure 8.1 and its GIMPLE version in Subfigure 8.2, immediately following the gimplification process (c.f., Section 2.2.2.1). The call to `strlen` has been optimized out, and the expression `strlen("hello" + i)` has been replaced by the expression `i < 5 ? 5 - i : 0`, much faster to execute and corresponding to the highlighted basic block `<bb2>` in Subfigure 8.2. Alas, when the out-of-bound checker analyzes the code in Subfigure 8.2, it is not aware that this code is, in fact, an out-of-bound access on the constant string `"hello"`.
- **Missing Optimizations**: warning emission passes usually rely on the result of other optimization passes, and in the absence of those results, false negatives might happen.
  Figure 10 illustrates it through the warning responsible for detecting `NULL` ptr passed as an argument to a function expecting a non-null argument. Two similar C functions are presented in Subfigure 8.1 and Subfigure 8.2. The only difference between them is the scope of variable `a`: it is global in Subfigure 8.1 and local in Subfigure 8.2. After the different machine-independent optimizations and before lowering to RTL, the `-Wnonnull` checker is executed. The GIMPLE representations of programs in Subfigure 8.1 and Subfigure 8.2 analyzed by this checker are, respectively, Subfigure 8.3 and Subfigure 8.4. The checker detects an issue in Subfigure 8.3 but not in Subfigure 8.4, mainly due to optimization passes having optimized out the global array `a` and replacing its usage in the call to `strcpy` by a `NULL` ptr in Subfigure 8.3. However, it is not the case when `a` is a local variable in Subfigure 8.4, because local variable are initialized dynamically.
  This effect and the fact that the `-Wnonnull` checker lacks deeper reasoning lead to the non-emission of the warning, despite being an obvious false negative.

2.5. Plugins

Similar to LLVM, the compilation process of GCC can be augmented through user-provided components called plugins. Plugins are shared object files passed through a command-line option to be loaded at runtime, and their scope is quite wide: they may implement a compiler pass for any purpose, whether it is for analysis or optimization, or add a new GCC attribute (e.g., type, function). This diversity is handled by defining different events represented through the `plugin_event enum`.

To be recognized and properly loaded, a plugin must implement a specific interface defined by the GCC plugin API: the `plugin_init` function. On GNU/Linux, the symbol is looked up in the shared object with `dlsym`. Upon success, the shared object implementation of `plugin_init` is called, enabling any initialization needed by the plugin. During this entry point execution, function callbacks can be registered for any event through a call to `register_callback`. Several events are defined by GCC, corresponding to different stages of the compilation pipeline. Here are non-exhaustive callback examples intervening at different stages of the pipeline:

- at the start or end of functions parsing for C and C++ frontends;
- during pragmas registration;
- before or after all passes execution;
- add an attribute to the attribute system;
- add a new analysis to the GSA (c.f., Chapter 4);

Once callbacks are registered, GCC will call them when, and if, their associated event occurs. The advantage of this approach is that it does not require recompiling GCC to hack around the compiler, its internals, or its pipeline. Figure 11 illustrates plugin registration to GCC.

```c
i = 17;
return strlen ("hello" + i);
```

Subfigure 9.1

```
i = 17;
_1 = (sizetype) i;
if (_1 <= 5) goto <D.3147>; else goto <D.3148>;
<D.3147>:
_2 = (sizetype) i;
D.3146 = 5 - _2;
goto <D.3149>;
<D.3148>:
D.3146 = 0;
<D.3149>:
return D.3146;
```

Subfigure 9.2

Figure 9: Example of a compiler's optimization introducing a fix to a vulnerable code from M. Sebor [66], even though it is an aggressive optimization of a C standard function [52, section 7.26.6.4].

```c
const char a[] = "123";
void f(char *d) {
  char *p = strchr(a, '9');
  strcpy(d, p);
}
```

Subfigure 10.1

```c
void g(char *d) {
  const char a[] = "234";
  char *p = strchr (a, '9');
  strcpy (d, p);
}
```

Subfigure 10.2

```
f (char * d) {
  <bb 2> [local count: 1073741824]:
  strcpy (d_2(D), 0B); [tail call]
  return;
}
```

Subfigure 10.3

```
g (char * d) {
  <bb 2> [local count: 1073741824]:
  a = "234";
  p_3 = strchr (&a, 57);   // 57 == '9'
  strcpy (d_4(D), p_3);
  a ={v} {CLOBBER};
  return;
}
```

Subfigure 10.4

Figure 10: Examples illustrated by M. Sebor [66] for the `-Wnonnull` warning [58, p. 110].

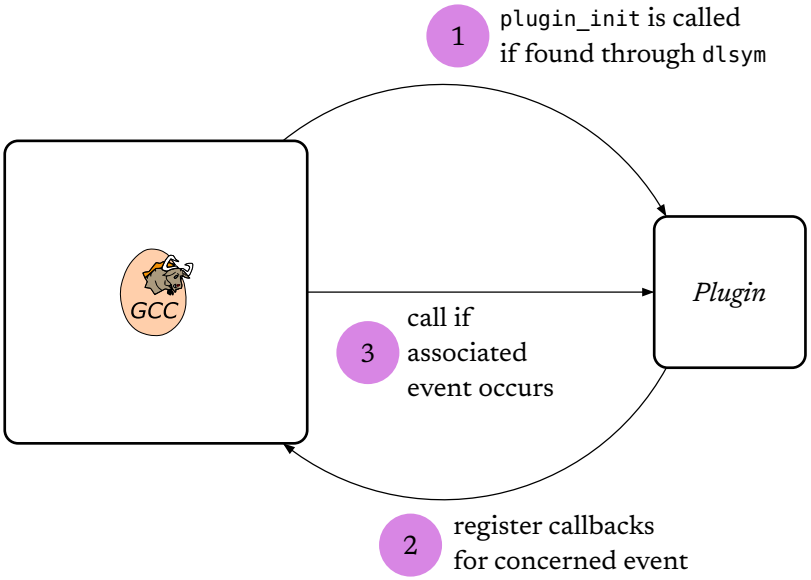


Figure 11: Relation between a plugin and GCC when invoked on Command Line Interface (CLI) with `-fplugin=/path/to/plugin.so`.

Chapter 3

Program Analysis

Program analysis is a cornerstone of modern software engineering, whether it aims at optimizing programs or at program security, by leveraging vulnerability detection during development stages, preventing it from being released vulnerable in the wild. Analysis techniques focus heavily on a memory model, whether it is for code optimization or for security-related analyses (e.g., buffer overflows, null pointer dereference, or use-after-free when considering C programs).

Still, programs may suffer from vulnerabilities tied to non-functional properties. For example, some programs have a notion of secret values (e.g., cryptographic keys or passwords), which is not tied to the source language itself, but to the program's purposes and internal semantics. Nonetheless, such analysis usually relies on the same core techniques as those targeting program safety.

Academic literature and industrial practices generally divide program analysis methods into three main categories: static, dynamic, and hybrid analysis. The chapter discusses each approach, enumerating some of their core techniques with sources from both academic research and industrial tools, with a strong focus on static analysis, which is of specific interest for this thesis. Section 3.1 introduces the notions of *soundness* and *completeness* in the context of program analysis. Section 3.2 discusses static analyses. Section 3.3 and Section 3.4, respectively, introduce dynamic and hybrid analyses.

Contents

3.1. Soundness vs. Completeness 42
3.2. Static Analysis 42
3.2.1. Symbolic Execution 42
3.2.2. Abstract Interpretation 44
3.2.3. Data-Flow Analysis 45
3.2.4. A Few More Techniques 47
3.3. Dynamic Analysis 47
3.4. Hybrid Analysis 48

Property	Effect	Possible False Negatives	Possible False Positives
Sound	Over-approximation	⊘	✓
Complete	Under-approximation	✓	⊘
Sound & Complete	Exact approximation	⊘	⊘

Table 2: Soundness and completeness relation and effect on analysis result.

3.1. Soundness vs. Completeness

In the context of program analysis, *soundness* and *completeness* are distinct and complementary concepts used to express the modelling properties of a given analysis.

Soundness

An analysis is *sound* if all possible behaviors of a program are represented in the analysis result, meaning the analysis never produces false negatives.

Definition 1

Completeness

An analysis is *complete* if it never reports alarms for situations that cannot occur in any real execution of a program, meaning no false positives should be reported by the analysis.

Definition 2

Achieving both soundness and completeness is infeasible for non-trivial properties of programs, due to Rice's theorem [67]. In recent years, B. Livshits *et al.* [68] introduced the notion of *soundiness* as a trade-off between soundness and precision. As defined by the authors, a *soundy* analysis tries to reach soundness by over-approximating specific language features while under-estimating others in some cases to scale to large codebases. Interestingly, they invite paper's authors involving soundy analysis to "*explain the general implications of their unsoundness and evaluate the implications*" on their evaluation instead of leaving "*sources of unsoundness [...] lurk in the shadows*". Table 2 illustrates the relationship between an analysis behavior and its possible outcome when considering those concepts.

3.2. Static Analysis

Static analysis inspects programs without executing them by using code inspection to find either optimizations that will leave the program semantics unchanged while reducing its execution time, or security weaknesses present in the code. By design, static analysis has no runtime overhead, but might increase compilation time when performed within compilers, and it ranges from simple pattern searches to advanced semantic analyses. For example, syntactic static analysis searches for suspicious code patterns (e.g., calls to unsafe C library functions). In contrast, semantic analysis constructs an abstract representation of the program (e.g., control-flow and data-flow graphs) to find deeper issues.

Static tools can operate on different representations of the program, whether it is on an intermediate representation [69–72] or an already compiled code [73]. In all cases, they often check for memory model-related issues. For example, when considering C/C++: array bounds, pointer misuses, integer overflows, type errors, and control-flow integrity violations. Nevertheless, some works also aim at non-functional properties such as program secret leaks or observable discrepancies based on execution time.

Several static analysis techniques exist, and undoubtedly, static analysis tools can implement several of them to analyze programs. This section is of specific interest for this thesis. It presents various static analysis techniques: symbolic execution in Section Section 3.2.1, abstract interpretation in Section Section 3.2.2, and dataflow-related techniques in Section Section 3.2.3. And finally, Section 3.2.4 introduces other static analysis techniques.

3.2.1. Symbolic Execution

Symbolic execution is a program-analysis technique where inputs are treated as symbolic variables instead of concrete values and originally comes from the program testing research area [74–78]. The program is then executed symbolically by following the program's CFGs. On each condition, the executor forks, following both paths and accumulating *path constraints* under a logical formula over the symbolic inputs that must hold for a given path.

```c
int foo(int x) {
  if (x > 0) {
    if (x == 42) {
      // special case or bug
    }
  }
  return 0;
}
```

Listing 7: Simple C code snippet of a function `foo` taking an `int` as parameter.

For example, on a conditional such as `if (sym > 0)`, execution forks into two states: one with the constraint `sym > 0` for the `true` branch, and with `sym ≤ 0` for the `false` one. In effect, the code is virtually executed, usually up to some arbitrary bound, for all possible inputs simultaneously. When an execution path ends, either by returning or calling an aborting function, an SMT solver (e.g., Z3 [79] or Boolector [80]) can be invoked to solve or reduce the path condition, potentially yielding a concrete input that will follow the same path in a concrete run. This behavior is illustrated in Listing 7.
Symbolic execution begins with parameter `x` being symbolic and unbounded. The first `if (x > 0)` on line 2 leads to two distinct paths: one with constraint `x > 0` and one with `x ≤ 0`. In the `c x > 0` branch, the second `if (x == 42)` on line 3 further splits into two other distinct paths with constraints `x > 0 ∧ x == 42` and `x > 0 ∧ x ≠ 42`. An SMT solver can then solve `x > 0 ∧ x == 42`, yielding `x == 42`, which means that the input `42` will drive the program through the innermost branch.

3.2.1.1. Limitations

Symbolic execution has well-known limitations:

- **Path Explosion**: Every conditional doubles the number of paths; programs with many branches or loops can result in an exponential number of paths, quickly becoming costly to explore fully, both in terms of time and memory [81,82]. This explosion can be mitigated with heuristics such as (1) merging paths if their inner states are equal, or (2) pruning paths during exploration if they are not feasible with respect to previously gathered constraints or if they reach an impossible state. Considering C, an impossible state could be reached by instructions leading to UBs, for example.
- **Memory model**: Memory model is a cornerstone of symbolic execution tools, as it will determine how the tool will handle some language-specific features [81,82]. Focusing on the C language, a memory model to represent memory objects is essential, as it will determine if a symbolic execution can properly handle complex C constructs. Memory objects in C may share the same storage, such as union-typed constructs that represent an "*overlapping nonempty set of member objects*" [52, §6.2.5]. Z. Xu *et al.* [83] have proposed a memory model based on the concept of region (c.f., Section 5.3).
- **Constraint Solving**: Path conditions can become very large or involve hard-to-solve theories (e.g., deep bitwise logic), causing solvers to time out or fail on constraint solving or reduction [81,82]. For example, when considering C, symbolic pointers, offsets, or indices might generate complex constraints hard to solve [81].
- **Environment Interaction**: Real programs usually invoke libraries or operating system APIs, but they might also rely on external data (e.g., files or network packets). C. Cadar *et al.* [84] mention this issue as "*the environment problem*". In a nutshell, symbolic execution must model or stub such behaviors; otherwise, missing or incorrect modelling might block or even wrong the analysis [82,84].

Some approaches also combine other techniques from static or dynamic analysis to mitigate the limitations of symbolic execution, as discussed in Section 3.3.

3.2.1.2. Symbolic Execution in Academic Works

Several tools from academic work embed symbolic execution implementation. C. Cadar *et al.* [84] introduce KLEE, which compiles C programs to LLVM bitcode and symbolically executes them to generate high-coverage test suites automatically. Their article reports testing GNU coreutils and other complex programs, achieving over 90% coverage and finding previously unknown bugs.
In more recent work, Y. Yuan *et al.* [85] enhances symbolic execution path's exploration by leveraging it with information from traces resulting from concrete calls to external library's functions.

3.2.2. Abstract Interpretation

Abstract interpretation, a research area in formal methods, has been theorized by P. Cousot *et al.* [86], and is a unified, mathematical framework for designing sound static analyses of programs. Its central insight is that the exact semantics of a program can be modelled as a fixpoint computation over a concrete domain of all possible program states. Since computing this fixpoint directly is often infeasible, abstract interpretation computes an approximate fixpoint over a smaller, abstract domain that retains enough information to prove desired properties [72,87].

3.2.2.1. Concrete and Abstract Domains

In abstract interpretation, the set of all possible program states forms the concrete domain. The abstract domain represents sets of concrete states using properties of interest (e.g., numeric intervals, possible signs of variables, aliasing information).

An abstract domain is organized as a lattice:

- **Top (`T`)**: the element at the very top of the lattice, which means "no information known".
  Typically, it is the initial state of every variable before the analysis begins. In a range analysis, it would be the interval `[MIN_INT, MAX_INT]`, representing all the possible values for an `int`.
- **Bottom ($\bot$)**: the element at the very bottom of the lattice, which represents an inconsistent or impossible state.
  For example, when considering the condition `if (x > 0 && x ≤ 0)`, as it is contradictory, the resulting state is $\bot$.
- **Partial order ($\leq$)**: operator capturing the partial order relation between two elements of the lattice.
  Suppose we consider the previously introduced elements $\top$ and $\bot$, then $\bot \leq \top$, because $\bot$ gives more information than $\top$ on the program's state.
- **Join ($\vee$)**: least upper bound, merges results from different paths.
  In a range analysis, for two states $[0, 1]$ and $[2, 2]$, the resulting state is $[0, 2]$. Or in a more mathematical style, $[0, 1] \vee [2, 2] = [0, 2]$.
- **Meet ($\wedge$)**: greatest lower bound, intersection of information.
  In a range analysis, for two states $[0, 1]$ and $[2, 2]$, the resulting state is $\bot$. Or in a more mathematical style, $[0, 1] \wedge [2, 2] = \bot$.

3.2.2.2. Abstraction and Concretization

Two functions connect the concrete and abstract domains:

- α: abstraction, it maps concrete states to an abstract element.
  In a range analysis, $\alpha(\{x = 4\}) = [4, 4]$. Or in other words, the function α applied to the concrete state $x = 4$, outputs the interval $[4, 4]$.
- γ: concretization, it maps an abstract element to the set of concrete states it represents.
  In a range analysis, $\gamma([2, 4]) = \{x = 2, x = 3, x = 4\}$. Or in other words, the function γ applied to the interval $[2, 4]$ from the abstract domain, outputs the set of concrete states $\{x = 2, x = 3, x = 4\}$.

If those two functions form a Galois connection, it ensures analysis soundness:

$$\alpha(c) \leq a \Leftrightarrow c \in \gamma(a)$$

The Galois connection guarantees that anything true in the concrete world is reflected in the abstract domain. In case of over-approximation in the abstract domain, false positives can still be emitted by such analysis.

3.2.2.3. Transfer Functions

Each statement of the program can be represented by a transfer function that describes how it changes concrete states. In abstract interpretation, abstract transfer functions are used to act over the abstract domain while preserving soundness (c.f., Figure 12).

3.2.2.4. Fixpoints

Within programs' CFG, loops are frequent. As the analysis interprets each statement's side effects, it might get stuck in an infinite analysis loop. To avoid this, the analysis computes a fixpoint, which is an abstract state that no longer changes upon new "execution" of the loop.

Transfer function:

$$f^c_{\text{inc}}(x) = x + 1$$

Current concrete state for x:

$$\{x = 4\}$$

Next state by applying f^c_{inc} to current state:

$$f^c_{\text{inc}}(\{x = 4\}) = \{x = 5\}$$

Subfigure 12.1

Abstract transfer function:

$$f^a_{\text{inc}}([l, h]) = [l + 1, h + 1]$$

Current abstract state for x:

$$x \in [0, 23]$$

Next state by applying f^a_{inc} to current state:

$$f^a_{\text{inc}}(x \in [0, 23]) = x \in [1, 24]$$

Subfigure 12.2

Figure 12: Transfer functions for the statement `x = x + 1` for the concrete domain in Subfigure 12.1 and the abstract domain in Subfigure 12.2 for a range analysis, alongside their application to example states.

Some fixpoints might be reached naively after a few interpretations of the body of convergent loops. Still, regarding divergent loops, a widening operation ∇ is performed on the abstract state to reach convergence faster, usually inducing over-approximation.

Analysis might recover some precision after a widening operation is applied on a fixpoint to limit over-approximation. To do so, it can use a narrowing operation Δ that will use any information available within the program's current abstract state set to regain precision on the fixpoint before ending its computation.

3.2.2.5. Abstract Interpretation in the Real World

Some tools based on abstract interpretation exist in the wild, not all originated from academic work, aside from the foundational work of P. Cousot *et al.* [86].

- ASTRÉE is a tool coming from academic work, historically designed for analyzing synchronous C programs [72].
- Frama-C, a code analysis framework embedding the EVA plugin, based on abstract interpretation [88].
- Infer is an industrial tool, designed for analyzing several source languages (including C) [70].

3.2.3. Data-Flow Analysis

Data-Flow Analysis (DFA) is a technique originally developed in compiler theory by G. A. Kildall [89]. Its goal is to compute information about the possible set of values calculated at various points in a program. DFA provides the foundation for constant propagation optimization, reaching definitions analysis, live variable analysis, or information flow tracking [14].

DFA associate abstract states (e.g., the set of variables that are definitely initialized or the possible value ranges of variables) to a program point (i.e., a node in the CFG). The analysis works by iteratively propagating these states along the CFG until a fixpoint is reached. A fixpoint in DFA is similar to the previously presented notion in Section 3.2.2.4: no changes are reflected on the program's state upon a new CFG walk. At each node, a flow function is applied to the node's entry state and generates its output state. In a nutshell, a flow function describes how statements transform the abstracted program state (very similar to abstract transfer function Section 3.2.2.3).

3.2.3.1. Reaching Definition Analysis

A reaching definition analysis, or *reach-def*, associates unique identifiers to program statements defining variables. When a variable is then used in a given statement, the analysis can identify which previous statements it depends on for its data. This analysis technique is broadly used within compilers for optimization purposes. A reach-def analysis of the program shown in Listing 9 would track which definitions of `x` may reach the following program points. The analysis is then able to tell that the `use(x)` statement on line 5 is dependent on the two definitions of `x` on lines 1 and 3. A constant propagation optimization pass could exploit this result to optimize the code by removing variable `x`, as illustrated in Listing 10.

3.2.3.2. Taint Analysis

Taint analysis [90] is an application of data-flow analysis in security, traditionally for the detection of unvalidated user inputs. Programs often take external data (e.g., from users or files). Suppose this data is used unchecked in some operations. In that case, it can lead to serious vulnerabilities such as SQL injection or buffer overflows [91–

```c
int x = 0; // def1
if (cond) {
  x = 1; // def2
}
use(x);
```

Listing 9: Simple C code snippet.

```c
if (cond) {
  use(1);
}
else {
  use(0);
}
```

Listing 10: Possible optimized version of Listing 9 resulting from a constant propagation optimization pass.

95]. The following concepts are commonly used within taint analyses and defined within the *SEI CERT C Coding Standard* [96].

Source

Represent the source of the taint. It might be the output of a function (e.g., `gets` for user's input or `rand` for random data) or a local variable.

Definition 3

Sink

Represents operations that should not be used with tainted data, otherwise leaving the code vulnerable. Depending on the analysis purposes, it might be any statement or expression (e.g., a function call or a pointer dereference).

Definition 4

Sanitization

A mechanism responsible for removing the taint on data. The function or operations destructing the taint are also called *destructors*. It corresponds to the flow function responsible for taint destruction.

Definition 5

Propagation

Mechanism responsible for implementing the taint's propagation semantics depending on the kind of statements. Functions and operations, also known as *propagators*, are analysis-dependent. It corresponds to the flow function for this kind of DFA.

Definition 6

Taint analysis is a data-flow technique where values originating from specified sources are marked as *tainted*. The analysis then propagates this taint through assignments and expressions. If tainted values reach security-sensitive operations (e.g., memory access or system calls) without passing through a validation or sanitization function, a potential vulnerability is reported (c.f., Figure 13).

Historically, taint analysis was used for user-input validation [91–94] but has also been used for tracking propagation of sensitive data within a program (e.g., password or cryptographic keys) [28,97].

3.2.3.3. Relation to Abstract Interpretation

DFA is relatively close to abstract interpretation:

- Flow functions are similar to abstract transfer functions: they are responsible for reflecting statement semantics on the program state's abstraction.
- The iteration to a fixpoint is the same lattice-based process, though traditionally framed in compiler theory for data-flow analyses rather than formal lattice semantics.

```c
void vulnerable() {
  char buf[100];
  int idx = get_user_input(); // idx is tainted
  if (buf[idx])
    do_stuff();
}
```

Subfigure 13.1

```c
void safe() {
  char buf[100];
  int idx = get_user_input(); // idx is tainted
  if (idx >= 0 && idx < 100) // idx gets bounded
    if (buf[idx])
      do_stuff();
}
```

Subfigure 13.2

Figure 13: Example of a C code vulnerable to buffer overflow Subfigure 13.1 and a version of it Subfigure 13.2 with a user-input validation, making it safe regarding this vulnerability.

One could see abstract interpretation as a formalization of DFA by providing a unified mathematical framework with domains, lattices, and Galois connections [86].

3.2.4. A Few More Techniques

Apart from the aforementioned static analysis techniques, here is a non-exhaustive list with a brief introduction of other methods used in static analysis:

- ***Pointer/Alias Analysis***: a program analysis technique establishing relationships between pointers and memory objects to statically assess to which objects a given pointer *might* point to or which pointers take the address of a given memory region. Several algorithm exists, for example: Andersen's [98], Steensgaard's [99] and Data Structure Analysis (DSA) [100].
  This analysis technique is common in compilers for code optimization. LLVM implements both Steensgaard's and DSA's algorithms, whereas GCC implements a points-to analysis inspired by a modified Andersen's algorithm [101].
- ***Model Checking***: a formal methods technique using a model (e.g., an automaton or a CFG) to represent every possible behavior of a program, through a set of states and transitions. This model is then exhaustively explored to try to prove that a specific property holds for this program [102].
  Several applications exist, including: the verification of concurrent programs by modelling them using *temporal-logic* checking [103], or verification of ANSI C programs based on *Bounded Model Checking* [104,105].
- ***Deductive Verification***: a formal methods technique using user-provided information and assertions about a program's functions to prove that they respect their provided assertions according to the properties and semantics associated with the programming language's instructions.
  For example, Frama-C with its WP plugin [69] or Rocq (previously known as Coq) [106] are proof-assistant systems based on deductive verification.

3.3. Dynamic Analysis

Dynamic analysis observes actual program behavior during execution, historically designed to measure programs' performance at runtime [107]. But factual memory errors in context can also be spotted by such analysis, as it focuses on actually executed paths in the analyzed program. Hence, it typically has a low false positive detection rate, albeit at the cost of increased runtime overhead.

Several approaches exist regarding programs' dynamic analysis. Here is a non-exhaustive list, including the main techniques:

- ***Code Instrumentation***: it is a technique used to wrap specific portions of the program with code checking the desired properties at runtime [108].

Since then, it has been adapted to detect security-related vulnerabilities or weaknesses, with some famous examples including:
  - *AdressSanitizer* [109] checking for invalid uses of pointers by implementing a shadow memory used to perform checks on the address used during execution (e.g., Null Pointer Dereference (NPD)).
  - *MemSan* [110] detecting use of both stack and heap-allocated uninitialized memory.
  - *Valgrind* [111], a tool instrumenting a program's memory to detect and debug memory-related issues (e.g., heap-allocated memory leaks).

- ***Fuzzing***: a technique of software testing, introduced by the work of B. P. Miller *et al.* [112], that mutates inputs of programs either through random input generation or guided input generation. This technique can be combined with symbolic execution to drive input generation to trigger specific paths upon execution (c.f., Section 3.4).
  A famous open source fuzzer is *AFL* [113,114], on top of which a lot of academic research has been done [115–117].

- ***Dynamic Taint Analysis***: it is the dynamic counterpart of the previously introduced static taint analysis in Section 3.2.3.2 [90]. It also relies on the notion of sources and sinks, but depends on concrete execution for the taint propagation.
  Secretgrind [118] is a dynamic analysis tool designed to track the propagation of secret data within a program, helping developers understand where secret data is copied. It also provides a report, stating if secret data remains in memory objects, potentially detecting non-zeroization of secret data. SOAAP [119] is a dynamic tool designed to help developers assess correct component compartmentation through user-provided information in the code. Both Secretgrind and SOAAP are based on Taintgrind [120], a dynamic taint propagation built on top of Valgrind's MemCheck [111].

3.4. Hybrid Analysis

Hybrid approaches combine static and dynamic analysis: static results can guide dynamic exploration, and reciprocally, dynamic results can refine static analysis. Hybrid analysis tools use techniques from both static and dynamic analysis to refine their analysis.

Here is a non-exhaustive list of such approaches:

- ***Concolic***: a contraction of *concrete* and *symbolic*, as this approach mixes concrete execution with symbolic reasoning for different purposes.
  For example, it has been used in program testing [121,122], specifically to tailor random input generation. In more recent work, S. Poeplau *et al.* [123] introduce SymCC, an LLVM-based compiler that embeds *concolic* execution (c.f. Section 3.4) into the binary itself. They utilize concrete execution to guide symbolic reasoning and report their approach as being significantly faster than traditional symbolic engines. Outside of academic work, the open source binary analysis tool *angr* [124] is also using techniques from both static and dynamic analysis to perform a wide variety of tasks, including automatic exploit generation, automatic Return-Oriented Programming (ROP) chain building, or automatic binary hardening (e.g., encrypting the return address of unsafe functions).

- ***Fuzzing + Static Analysis***: Several academic works relied on static analysis to enhance fuzzing. Here is a non-exhaustive list:
  - N. Stephens *et al.* [125] uses symbolic execution when fuzzing programs to direct input generation to satisfy specific path constraints.
  - A. Mantovani *et al.* [126] relies on the Data-Dependency Graph (DFG) of programs to reward the fuzzer upon new paths explored in the DFG.

Part II

Contributions

Chapter 4

GCC Static Analyzer

In May 2020, alongside GCC 10, the first version of its internal static analyzer was introduced. This newcomer in the suite is a read-only pass, meaning that no modifications should be performed on the code. The GCC Static Analyzer (GSA) is enhancing the compiler's ability to detect issues in the code at compile-time, and consists of a framework that leverages GCC's internal structures and classes with symbolic execution (c.f., Section 3.2.1), enabling it to perform virtually any static code analysis at compile-time.

The GSA intervenes in the late stages of the middle-end, specifically on the late stages of GIMPLE SSA (c.f., Section 2.2.2.2). The main advantage of this position in the pipeline is that the code has already been through many optimization phases; therefore, performing an optimization-dependent analysis to detect issues introduced or suppressed by the compiler. In contrast, as confirmed by the LLVM project in an issue on the project's GitHub [127], the Clang Static Analyzer (CSA) has no means to act on the Clang IR after optimization.

This chapter introduces the first contribution of this thesis: the dissection of the GCC Static Analyzer's internal mechanisms and data structures. Section 4.1 gives some context to the chapter. Section 4.3 and Section 4.4 respectively present the GSA's memory model and its program state representation. Section 4.5 dives into its mechanism responsible for analysis implementations, and Section 4.6 exhibits its reporting system. Section 4.7 present the relationship between the GSA and the GNU attribute system. Finally, Section 4.9 aims to assess the GSA's usability through an empirical evaluation.

Contents

4.1. Context and Motivations 52
4.2. Exploded Supergraph 52
4.3. Region-based Memory Model 53
4.3.1. Regions 55
4.3.2. Svalues 57
4.4. Program State 57
4.4.1. Store Mechanism 58
4.4.2. Constraint Manager 58
4.5. State Machine 58
4.5.1. Registration Through GCC's Plugin API 61
4.6. Reporting System 61
4.7. GNU Attributes System and GSA Relationship 62
4.8. Native GSA's warning 63
4.9. GSA vs. the World 66
4.9.1. Juliet Testsuite 66
4.9.2. Real-World vulnerability 67
4.10. Conclusion 67

4.1. Context and Motivations

As presented in Section 3.2, program static analysis is a prolific research area with numerous subdomains. However, different surveys focusing on developers' prism exist: B. Johnson *et al.* [18] focuses on static analysis tools, E. B. Sørensen *et al.* [128] aims at security-oriented program analysis and M. Christakis *et al.* [19] centers on the developers needs and wishes. Alongside this survey, feedback on building static analysis tools has been published by both researchers [129] and companies [20]. The results of those publications highlight several key points for developers to decide the (non-)adoption of static analysis tools. In a nutshell:

- **Analysis Results**: A false positive rate too high in a tool's result understandably leads to developers ignoring the tool (e.g., 30% rate for A. Bessey *et al.* [129]). However, an insight from developers highlights that presenting results in a user-friendly manner could reduce the false positive burden [18]. Another aspect is the warning intelligibility: a warning without any or with poor explanation tends to be ignored and refrain tool from adoption [18,19].
- **Development Workflow**: Tools should not break the developers' workflow: if developers need to change their development habits to use a tool, it tends to demotivate its adoption [18–20]. Both surveys and feedbacks illustrate the importance of integration into both Integrated Development Environments (IDEs) and the compiler [18,20,129].
- **Customization**: In their survey, B. Johnson *et al.* [18] illustrates that most interviewed developers attach importance to static analysis tools customization without requiring deep knowledge of their internals and to share them easily in a collaborative development environment. Several compatible mechanisms might leverage a tool's customization: 1) importable and exportable configuration file, 2) code or comment annotation to gain additional information on the codebase, or 3) user support to assist them in grasping the tool. Interestingly, E. B. Sørensen *et al.* [128] shows a weak preference for annotation-based tools over automatic tools, which is correlated by the results from M. Christakis *et al.* [19].

Considering those different developers' concerns, the GSA seems to be a suitable candidate: integrated directly in the compiler and IDEs through the Static Analysis Results Interchange Format (SARIF) format[3], alongside access to the GNU attribute system (c.f., Section 2.3) for customization.

The GSA, unlike GCC (c.f., Chapter 2), is entirely written in C++ and relies on a heavy abstract design based on C++'s structure and class hierarchy. Many internally handled components are abstract classes and structures, building a solid foundation for an entirely abstract static analyzer that adapts to various analysis classes.

Despite its promising potential, the GSA is a complex framework not used in practice in the literature, even through plugins. Two possible reasons are: 1) the GSA is ill-documented, despite ongoing efforts to improve its internal documentation, and 2) it "is neither sound nor complete" [17].

The GSA has no pretension of surrogating the existing GCC's warning engine, which is limited by design (c.f., Section 2.4). It is rather to provide a path-sensitive static analysis framework, powered by symbolic execution mechanisms, to enhance GCC's ability to analyze programs at compile time.

4.2. Exploded Supergraph

> **Takeaway:** The GSA represents programs through an extension of its CFGs, by adding interprocedural edges and duplicating nodes coming from different control-flow instructions. This mechanism allows it to analyze the same node in different contexts.

Firstly, the GSA generates the supergraph of the current translation unit by adding interprocedural edges (i.e., calls and returns) to the CFG of the different functions. Then, an exploded version of the supergraph is built, as illustrated in Figure 14. The work of T. W. Reps *et al.* [130], introducing the *Interprocedural, Finite, Distributive, Subset* (IFDS) framework, inspires the idea of the exploded supergraph in the GSA.

3. https://sarifweb.azurewebsites.net/

Reminder: The IFDS is a dataflow framework for analyses in which the set of facts is finite, the program state at each point is represented as a subset of these facts, and the transfer functions are distributive over the meet operator. Hence, extending intraprocedural analysis to interprocedural settings by handling function calls and returns. T. W. Reps *et al.* [130] reduces IFDS problems to graph reachability, leveraging the framework to guarantee polynomial-time solving while preserving precision.

Let's consider taint analysis (Section 3.2.3.2), which naturally fits into the IFDS setting:
- **Finite**: the set of facts is the finite collection of the program's variables.
- **Subset**: the analysis state is the subset of variables currently marked as tainted.
- **Distributive**: when different paths merge, the tainted sets can be combined by union.
- **Interprocedural**: taint information propagates across function calls and returns.

Combined with symbolic execution (c.f., Section 3.2.1), the exploded supergraph enables the analysis of the same block with possibly different incoming paths and different output paths while still being able to distinguish every path. Indeed, each exploded node is associated with a pair of a program point in the supergraph (not the exploded one) and a program state that holds interprocedural information (e.g., variables and function frames) and analysis states (c.f., Section 4.5). Each exploded node has a single predecessor and a single successor, implying that it retains virtually the old program state from its predecessor and the new program state passed to its successor, reflecting the path-sensitive symbolic evaluation of the current exploded node. The only exceptions regard:

- The entry node of a function can have several input edges up to the number of call instructions for that function in the translation unit, or none if it is never called.
- Following the same logic, function exit nodes can have several exit edges up to the number of call statements to this function within the translation unit, or none if it is never called.
- The exploding nodes containing condition control-flow statement (i.e., `if` or `switch`) can output several program states, one for each possible result of the statement. In practice, an `if` statement always has two possible out outcomes: its condition is either `true` or `false`. Hence, respectively leading to the execution of either: 1) its first substatement, or 2) its second substatement in the `else` form or its following statement otherwise. However, for `switch` statements, there are at least two output nodes. An exploded node containing a conditional control-flow statement may have at least two output nodes, each with a different program state depending on the value of the condition (c.f., Section 4.4.2).

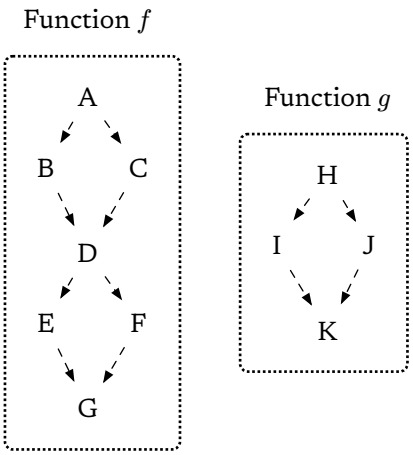
One of the drawbacks of this technique is the explosion in memory usage resulting from the large number of paths in a massive codebase. Hence, some cost reduction is implemented to reduce computation by potentially merging two exploded nodes if their associated program points and states are considered equal by the GSA.

Once the exploded graph is built, the GSA plans the analysis of each defined function in the translation unit, *as if* it were the entry point of the analysis, meaning that all defined functions in the translation unit will be analyzed at least once, without any calling context. If we consider the exploded graph from Figure 14, in which Subfigure 13.1 represents each function CFG, Subfigure 13.2 is the supergraph with call and return edge added, and finally Subfigure 13.3 shows the exploded supergraph of Subfigure 13.2 built by the GSA. Then, the GSA will consider two start points for its analysis: the entry point of both functions g and f. Hence, function g will be analyzed two times: once with the context of the call within function f when f is the entry point of the analysis, and a second time without any context by starting the analysis on function g.
This analysis plan leverages the ability to analyze the same functions in several and potentially different execution contexts.

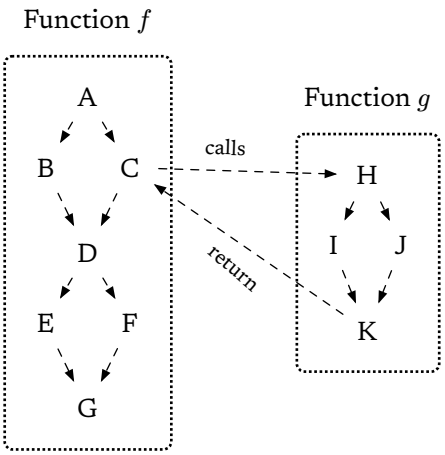
4.3. Region-based Memory Model

Takeaway: The memory model of the GSA is based on the work from C. S. Strachey [131]. To summarize, this region-based memory model is tailored to C programs and can represent the plurality of memory object constructs allowed by the C standard. Each memory object is represented by a `(region, svalue)` pair, with `region` representing the memory object's l-value, and `svalue` representing their r-value.

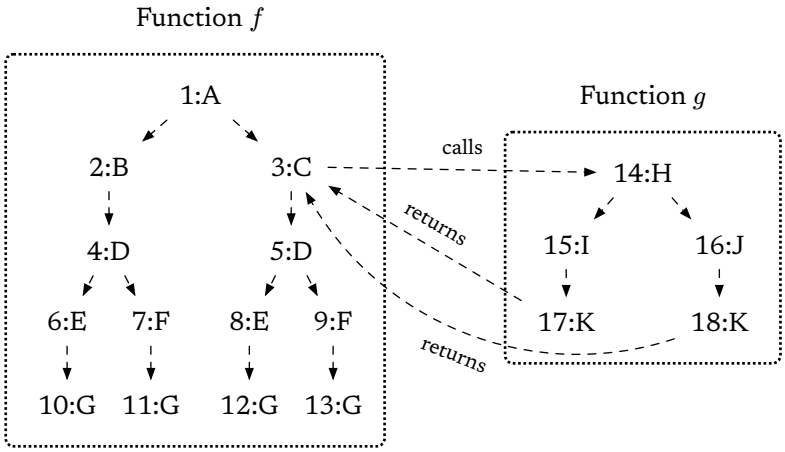
4. Release planned alongside GCC 16 in 2026.



Subfigure 14.1



Subfigure 14.2



Subfigure 14.3

Figure 14: Construction of the exploded graph of a given translation unit containing two functions f and g, with f calling g in its node C.

The GSA has been designed to analyze C code, although some work is currently being done to leverage thorough analysis of C++ code[4]. In programming languages theory, variables are usually represented as a pair of two components: an `l-value` and an `r-value`, defined as follows by C. S. Strachey [131].

`l-value`

An area of the store of the computer, or a *location*, of an arbitrary size that may or may not be addressable and with a content.

Definition 7

`r-value`

The content of an `l-value`.

Definition 8

The memory model within the GSA is heavily inspired by the work of Z. Xu *et al.* [83]. `L-values` are associated with the notion of `region`, including ones without identifiers (e.g., an element of an array, a function body, or even a stack frame). As for `r-values`, they are associated with the notion of symbolic values called `svalues` in the GSA (e.g., a constant value `42` or the address of a specific region).

This section will delve into the GSA's data structures used to represent a program memory model. Section 4.3.1 and Section 4.3.2 respectively dive into the `region` and `svalue` classes.

Space Region Class	Equivalent Binary Section	Description
`code_region`	`.text`	The `.code` binary section holds the executable code of a binary. It is represented within the GSA by the `code_region` class, a unique memory region that holds information about every function defined in the current translation unit, including static functions.
`globals_region`	`.data` `.bss`	The `globals_region` class is used by the GSA to represent a unique memory region that holds every location relative to global data of a translation unit, i.e., extern, static, and/or const global variables.
`function_region`	*N/A*	Represent the memory area containing the actual code of a function from the analyzed translation unit. All of them are contained within the aforementioned `code_region`. Its equivalent in the binary would be the exposed symbol for non-static functions.

Table 3: Equivalence between the binary sections and space regions classes used by the GSA to model a program's memory.

4.3.1. Regions

Within the GSA, regions are all based on a single abstract class called `region`, on top of which different kinds of concrete classes are implemented for two distinct purposes: 1) *space region* destined to represent an untyped memory area potentially containing other regions, and 2) *concrete region* aiming to represent actual locations.

A parenting hierarchy exists among the different regions. The `root_region` at the top is essentially representing the whole program's memory and encloses several other space regions to discriminate amongst different memory areas existing either in the binary (c.f., Table 3) or in memory at runtime (c.f., Table 4). The different space regions are intended to encompass several other classes representing various types of concrete memory locations, such as variables, array elements, or specific fields within a given structure instance (c.f., Table 5). The parenting relationship between different regions is illustrated in Figure 15.

Space Region Class	Description
`heap_region`	Unique memory region representing the heap's memory area of a program.
`stack_region`	Unique memory region representing the stack's memory area of a program.
`frame_region`	Represents the memory area of the frame of a function's stack, and its lifetime spans from the beginning to the end of its associated function's symbolic execution. All of them are contained within the `stack_region`. This class enables the GSA to distinguish between calls of the same function but with different call contexts, such as recursive functions. Thus, local variables are associated with a `frame_region` instance, thereby enabling context-related analysis.
`alloca_region`	Represents a memory area region dynamically allocated on the stack via a call to `alloca`. Each instance is associated with a unique `frame_region` and its lifetime is linked with that of its parent frame.
`heap_allocated_region`	Represents a memory area region dynamically allocated on the heap (e.g., through `malloc` or `calloc`).

Table 4: Space region classes used by the GSA to represent a program's runtime memory areas.

Concrete Region Class	Description
`decl_region`	Represents the typed memory location of a specific variable and is central within the analyzer internals. It contains a reference to the internal `tree` representation, which can be either a `VAR_DECL` for local and global variables, an `SSA_NAME` for SSA variables, or a `PARM_DECL` for a function's parameters. Whether it is global, local, an SSA variable, or a function's parameter, its lifetime is determined by its first-degree parent region: 1) `frame_region` for local, SSA variables and parameters, or 2) `global_region` for global variables.
`field_region`	Represents a field within a structure or a union. It is always associated with a type and a `decl_region` as its first-degree parent.
`element_region`	Represents an element within an array with its corresponding typed offset. Its first-degree parent region is the `decl_region` corresponding to the indexed array, and its type is the type of its parent's element.
`offset_region`	Represents a byte offset within another region with a typed offset and is used for pointer arithmetics. Its first-degree parent region is the base pointer.
`symbolic_region`	Represents a symbolic region. Such a region can be obtained by dereferencing an unknown non-null pointer, e.g., a parameter of a function without any call context.
`cast_region`	Represents a view of another typed region with a different type.

Table 5: A non-exhaustive table presenting other kinds of regions used internally by the GSA to represent a program's state.

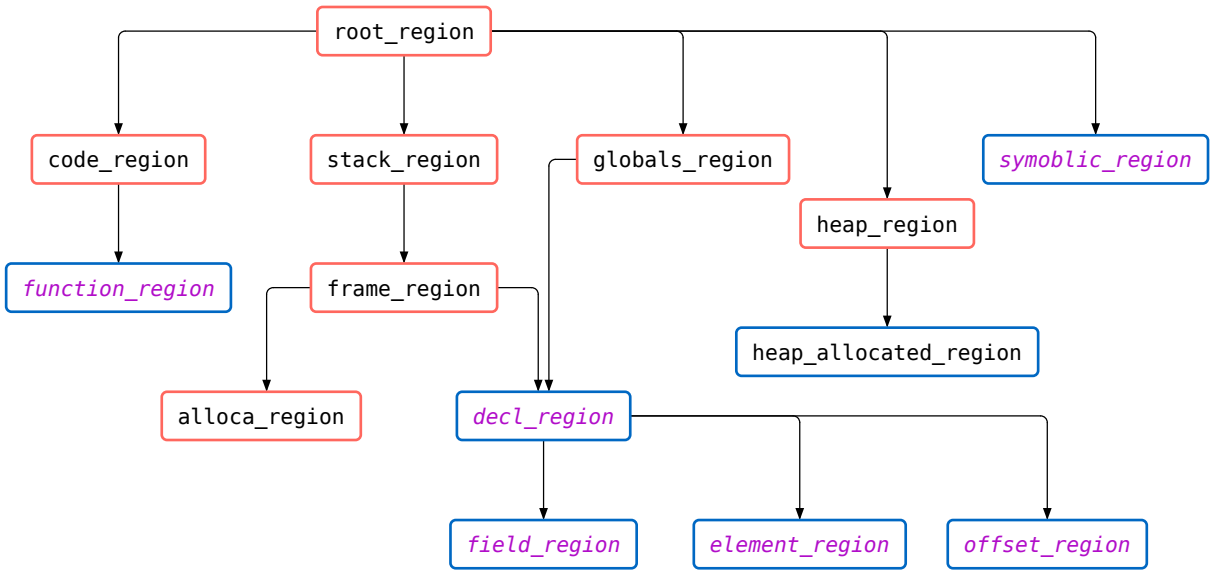


Figure 15: A Directed Acyclic Graph (DAG) representing the parenting relationship between the different regions implemented by the GSA, with typed region *highlighted* as those are the primary regions used during an analysis. The red nodes represent space region and the blue ones represent concretes region.
Note: the `cast_region` has not been represented here as it essentially serves as a wrapper around any typed region.

4.3.2. Svalues

To represent svalues, the GSA relies on an abstract base class `svalue`. Several concrete classes inherit it to represent different kinds of svalues. Two groups of svalues can be defined: 1) the self-sufficient ones, meaning they do not depend on any other svalues, such as a constant value or the address of a region (e.g., a pointer to a memory location), and 2) the one dependent on other svalues, such as binary operations (e.g., pointer arithmetics). Table 6 illustrates different kinds of svalues used internally by the GSA to execute a program symbolically.

Each svalue is typed, implying that two svalues will result in the same object if, and only if, they 1) have the same type, 2) are instances of the same concrete class, and 3) their internal svalues, if any, are the same. Let's consider two constant svalues, `constant_svalue(type: int, value: 42)` and `constant_svalue(type: char, value: 42)`, which will result in two distinct memory objects. Likewise, `unknown_svalue(type: int)` and `unknown_svalue(type: char)` are distinct objects. Reciprocally, if the svalue `constant_svalue(type: int, value: 41434142)` has already been created during the program analysis, no other one will be instantiated, so every place where this exact svalue is needed will point to the same memory object.

Svalue Class	Description
`constant_svalue`	Represents a specific constant value (e.g., 42).
`region_svalue`	Represents a pointer to a known concrete region.
`poisoned_svalue`	Represents a value that should not be used (e.g., the content of an uninitialized concrete region). This svalue is a per-type singleton, with respect to the program state.
`unknown_svalue`	Represents an unknowable value, the $\perp$ value when picturing the svalues as a lattice (c.f., Section 3.2.2.1). This svalue is a per-type singleton, with respect to the program state.
`initial_svalue`	Represents an initial value at the beginning of the analysis path and is linked to the `decl_region` it relates to, ensuring unicity for each `initial_svalue`. For example, this will be the svalue associated with the parameters of the entry function and the external global variables at the beginning of the analysis.
`unaryop_svalue`	A unary operation on another svalue, such as casting.
`binop_svalue`	A binary operation on two other svalues. For example, a pointer arithmetic operation can operate on a `region_svalue` (i.e., the pointer) and a `constant_svalue` (i.e., the offset).
`conjured_svalue`	This class represents an svalue that depends on a given statement, such as a specific function call. For example, an undefined function `f` is called with the pointer to a variable `a`, so the statement `f(&a)` potentially modifies the svalue associated with `a`. The reason `unknown_svalue` is not used here is to conserve precision in the analysis; it can be translated to "we don't know anything about this svalue besides its dependence on a call to the function `f` at this statement of the program". If the called function is defined in the translation unit, the GSA will analyze it with the calling context. Upon the callee's return, the svalue can be updated if the analysis has reached a more precise svalue.

Table 6: Non-exhaustive table of the different kinds of svalue concrete classes used by the GSA.

4.4. Program State

Takeaway: The GSA associates a program state to each node of the the exploded supergraph, accumulating several information regarding the program's state such as the different constraints leading to that node, its associated region model, alongside the state of the different analyses performed by the GSA.

As mentioned in Section 4.2, each exploded node is associated with a program state that holds path-sensitive information such as the current svalues for declared variables and the set of constraints gathered along its associated node's incoming path.

This section will inspect the different classes used to refine the program's state precision.

4.4.1. Store Mechanism

> **Takeaway:** The `store` data structure is mapping the regions to svalues. It is the core implementation of the region-based memory model.

A data structure is implemented to map regions to an svalue: the `store` class. To reduce memory overhead, the store does not store actual objects, but pointers to those objects. Implementation-wise, the store is a hashmap mapping `region` pointers to `svalue` pointers. Each program state has its own associated store, allowing it to reflect statement side-effects during symbolic execution. Since a region can have the address of another region as an svalue (i.e., pointers), the GSA comes with a free points-to analysis. Figure 16 illustrates this model over a C code snippet, where each variable is associated with a region and an svalue, wrapped in its corresponding `frame_region` for clarity. In practice, the frame regions are not directly saved in the store, but referenced directly in `decl_regions`. The svalue's kind `conjured_svalue` was mentioned earlier in Section 4.3.2, and it is illustrated in Subfigure 15.3. Moreover, the "free" points-to analysis is illustrated in the store with function `f`'s local variables `ptr` and `ptr_b` `decl_region` having the same associated svalue: `region_svalue(&x)`, the address of the `decl_region` of variable `x`.

4.4.2. Constraint Manager

> **Takeaway:** The constraint manager is the data structure used internally by the GSA to record the different constraints during path exploration when conditional control-flow instructions are met. It is a central data structure within the GSA, as it is used to determine if a given path is feasible.

During the exploration of the exploded graph, the GSA gathers information about conditional control-flow statements (e.g., `if` statements) into a constraint manager within the program state. The `constraint_manager` class implements this feature, essentially managing the set of constraints collected along the path.

As explained in Section 4.2, exploded nodes containing conditional control-flow instructions may have several successors. For each successor, upon entry, the GSA will record within the constraint manager the constraint over the svalues required to hold for entering it. Figure 17 illustrates this behavior by following a simple C code snippet.

At every program point, it is possible to ask the GSA to evaluate a condition of the form `lhs op rhs`, where `op` $\in \{<, >, \leq, \geq, ==, \neq\}$. The evaluation returns an instance of the `tristate` data structure representing the result of the condition's evaluation considering the current constraint manager's state, and it is either known (i.e. `true` or `false`) or unknown. So far, the GSA relies on heuristics to evaluate conditions. Still, leveraging it with SMT solvers (e.g., Z3 [79]) would be a highly interesting contribution, especially for computing path feasibility (c.f., Section 4.6).

4.5. State Machine

> **Takeaway:** This data structure is the heart of advanced analyses within the GSA. In a nutshell, the GSA enables state machines to respond to the program's statements and reflect potential side effects on their analysis within the program state.

The GSA performs checks that are categorized into "warnings". The native set of warnings covers a variety of analyses targeted at finding security and API usage bugs, dead code, and other logic errors [132]. The GSA itself is not performing any advanced analysis beyond detecting uninitialized value usage, thanks to the `poisoned_svalue`. Instead, to perform advanced analysis, the GSA relies on the `state_machine` abstract base class.

Before diving into the state machine internals, let us consider a GSA's native state machine that tracks the allocation state of a given pointer allocated through `malloc`: the `malloc_state_machine` (c.f. Figure 18). This state machine allows the GSA to detect misuse of allocated pointers, such as Use-After Free (UAF) (c.f. Listing 11). In practice, alongside tracking an svalue's allocation state, it also tracks a list of suitable deallocator function(s) (e.g., `malloc`/`free` or `new`/`delete`), leveraging to detect when an improper deallocator is called on an allocated object (e.g., `free` an object allocated with the `new` operator or through a call to `alloca`).
Interestingly, this mismatching deallocator detection feature can be augmented to custom allocator/deallocator functions thanks to the `[[malloc (custom_dealloc)]]` GNU's function attribute.

The concrete classes inheriting the `state_machine` class will react to specific events occurring during symbolic execution and modify their associated `sm_state_map` data structure within the program state accordingly, based on

```c
1  void f(void) {
2    int x = 42;
3    int *ptr = &x;
4    int *ptr_b = ptr;
5    int y = g(x);
6  }
7
8  int g(int y) {
9    int x = y + 624;
10   return x;
11 }
```

Subfigure 16.1

store		
	region	svalue
frame_region(f)	decl_region(x)	constant_svalue(42)
	decl_region(ptr)	region_svalue(&x)
	decl_region(ptr_b)	region_svalue(&x)

Subfigure 16.2
Store corresponding to the program state after the symbolic evaluation of line 4, in function `f`.

store		
	region	svalue
frame_region(f)	decl_region(x)	constant_svalue(42)
	decl_region(ptr)	region_svalue(&x)
	decl_region(ptr_b)	region_svalue(&x)
	decl_region(y)	conjured_svalue (int y = g(x))
frame_region(g)	decl_region(y)	constant_svalue(42)
	decl_region(x)	constant_svalue(666)

Subfigure 16.3
Store corresponding to the program state after the symbolic evaluation of line 10, in function `g` *before* returning to function `f`.

store		
	region	svalue
frame_region(f)	decl_region(x)	constant_svalue(42)
	decl_region(ptr)	region_svalue(&x)
	decl_region(ptr_b)	region_svalue(&x)
	decl_region(y)	constant_svalue(666)

Subfigure 16.4
Store corresponding to the program state after the symbolic evaluation of line 5, in function `f` *after* returning from function `g`.

Figure 16: The relationship between regions and svalues within the store mechanism, based on an analysis of the C code presented in Subfigure 16.1.

their analysis. The `sm_state_map` is mapping svalues to a pair consisting of an arbitrary state and its origin (i.e., the svalue's "responsible" for that state, if any). A state, implemented in the `state` concrete class, is arbitrary, as it is analysis-dependent and meaningful only to its state machine. The `state_machine` class contains a default `start` state, which is the state of every variable not tracked within its corresponding `sm_state_map`. This `state` class is extendable through C++'s inheritance mechanism if the state machine needs to add more information within one of its states (e.g., the deallocator information for the malloc state machine). The origin information may be useful when building a path to investigate an issue for further analysis. To the best of our knowledge, none of the native state machines in the dissected GCC version [35] uses the origin information. Reasons are twofold. First, analyses are mostly unrelated to the svalues themselves, but rather on their logical state (e.g., allocated, freed). Secondly, state machines rely on another mechanism being part of the reporting system (c.f., Section 4.6).
The GSA directly handles the different state maps with per-state machine uniqueness. Hence, the state machines are not responsible for managing their state map, as the GSA is handling it for them by linking them to a program state, and thus to an exploded node.

To reformulate, state machines "only" react to an event by transitioning a previously computed state into a new one reflecting the event's potential side-effect(s). To that end, the `state_machine` abstract class provides a set of

```c
void f (int x) {
  int y = 42;
  if (x >= y) {
    // do something
  }
  else {
    // do something else
  }
}
```

Subfigure 17.1

store

region	svalue
decl_region(x)	initial_svalue(x)
decl_region(y)	constant_svalue(42)

constraint_manager

initial_svalue(x) $\geq$ constant_svalue(42)

Subfigure 17.2

Program state's store and constraint manager upon entry of block starting on line 4 of Subfigure 17.1, i.e., the `true` branch.

store

region	svalue
decl_region(x)	initial_svalue(x)
decl_region(y)	constant_svalue(42)

constraint_manager

initial_svalue(x) $<$ constant_svalue(42)

Subfigure 17.3

Program state's store and constraint manager upon entry of block starting on line 7 of Subfigure 17.1, i.e., the `false` branch.

Figure 17: The constraint manager feature, alongside the store, for clarity regarding the different possible exit edges of Subfigure 17.1 when considering the condition on line 3. The only difference between the program states Subfigure 17.2 and Subfigure 17.3 is the operator on the relation.

virtual methods that are called at various points during the program's symbolic execution. Most of these methods take as a parameter an instance of the `sm_context` class, which provides a view of the current analysis state tailored to a specific state machine. This data structure has read-only access to the old program state and complete access to the new program state of the current exploded node, alongside pointers to both its old and new state maps. The old state map corresponds to the state machine's analysis intermediary results before the evaluation of the current event. In contrast, the new one will be modified to reflect the event's side effects on the state machine's analysis, if any.

The central virtual method is `state_machine::on_stmt`, allowing a state machine to reflect the side-effect of a statement on its state map, and takes as parameters:

- A pointer to its `sm_context` instance.
- A pointer to the GIMPLE statement on which to react.
- A pointer to the current node of the supergraph, essentially for diagnostic purposes.

Here is a non-exhaustive list of other state machine's methods of interest:

- `on_phi`: allow to react on a GIMPLE Φ statement (c.f., Section 2.2.2.2).
- `on_pop_frame`: allow to react upon the return of a function, not the callee return statement, but the actual popping stack mechanism.
- `on_leak`: allow to react when a tracked `svalue` is not associated with any region within the store.

Interestingly, any analysis that can be modelled as a state machine is implementable, requiring only the necessary virtual functions to be implemented and the corresponding states for its state map.

The `sm_context` class allows the state machine to ask the GSA to potentially emit a warning at any program point if its state map reflects a state triggering an issue. For example, when the malloc state machine reacts to a call statement to `free`, if its argument is already in the `freed` state, a double-free is detected, and a warning is emitted accordingly.

```c
int *ptr = (int *) malloc(sizeof(int));
// ptr svalue state: start -> unchecked
free (ptr);
// ptr svalue state: unchecked -> freed
*ptr = 42;
// UAF as ptr svalue state is freed
```

Listing 11: A simple C code snippet presenting a UAF.

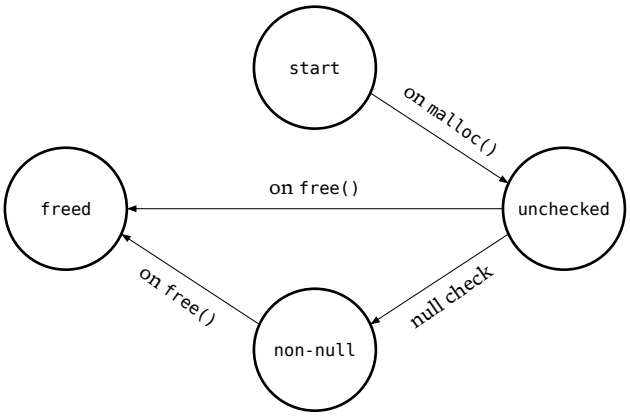


Figure 18: Simplified malloc state machine representation. For clarity, some transition edges have been removed, such as calls to `realloc`, `alloca`, or C++'s `new` operator.

4.5.1. Registration Through GCC's Plugin API

The GSA extended the GCC's plugin API to allow plugins to register a new state machine with the analyzer. Hence, extending the GSA's native set of warnings with new ones. For this, the imported plugin should implement a concrete class that inherits from the `state_machine` abstract class (c.f., Section 4.5) and register it with the GSA by registering a handler for the `PLUGIN_ANALYZER_INIT` event. The GSA will call this handler during its startup, which should register an instance of its state machine by using the handler's parameter. Once a state machine is registered, the GSA does not distinguish between its native state machines and those imported by the plugin. Thus, it allows new analyses to be performed by the GSA without the need to recompile GCC. This registration mechanism is illustrated in Figure 19.

4.6. Reporting System

Takeaway: The GSA's reporting system is one of its key components. It only intervenes in cases where warnings are waiting to be emitted. At which point, the reporting system will leverage the aforementioned constraint manager to compute path feasibility for the different warnings awaiting. Path feasibility computation can be costly; thus, the GSA does it lazily, meaning that no calculation is performed if no warnings are produced.

As a significant part of the GSA, this component comprises classes that report issues from the line, which then trace them back to their source. The highest-level class is the `diagnostic_manager`, which is responsible for determining if a given diagnostic should be emitted by trying to find, for each diagnostic, the shortest and feasible path within the exploded graph leading to it, regarding the associated context. To determine a path's feasibility, the GSA relies on heuristics and the set of constraints accumulated in the constraint manager alongside its exploration(c.f., Section 4.4.2). There are several outcomes to this pathfinding:

1. No feasible path is found, hence the considered diagnostic is discarded.
2. A feasibility limit is reached, which helps overcome the termination problem, and also discards the diagnostic.
3. Or at last, a feasible path is found, leading to the diagnostic being emitted (c.f., Image 1).

So far, the GSA is relying on heuristics to determine path feasibility. If an expression or a set of expressions is too complex, the path is considered feasible by default, which may result in a false-positive warning being emitted. An ongoing project within the GSA is to implement automatic queries to SMT solvers installed on the system when heuristics fail, if any.

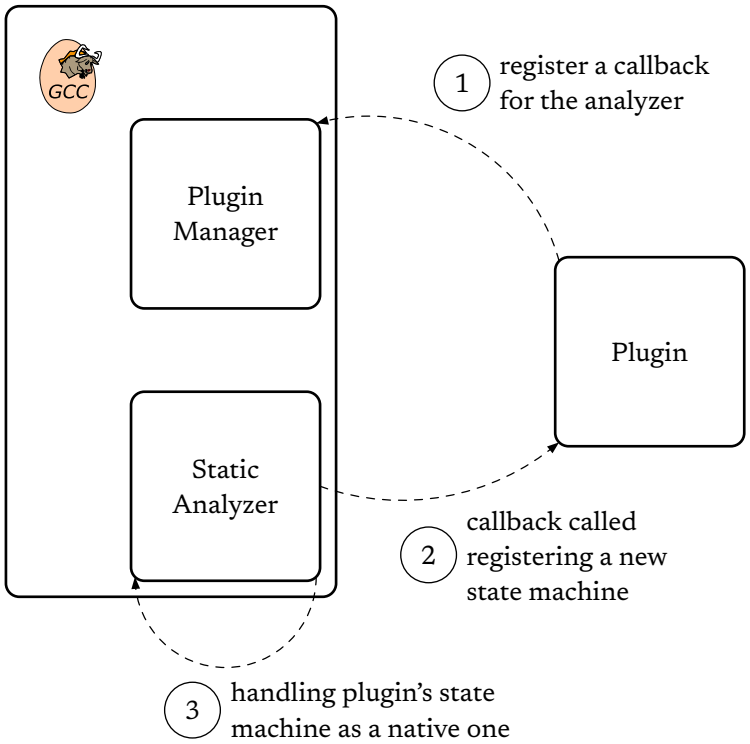


Figure 19: GCC/GSA plugin's registration workflow.
Invoked with: `gcc -fanalyzer -fplugin=/path/to/plugin.so input.c`

Finding the shortest feasible path is a high cost in time. Indeed, the whole set of paths leading to the precise node of the exploded supergraph triggering the warning needs to be looked at for both their feasibility and their length. Intuitively, the time needed to try to find the shortest feasible path is tied to the number of paths and their complexities (e.g., more constraints to satisfy).

Each state machine can implement its own diagnostic class by inheriting from the `pending_diagnostic` abstract class, which can then be registered with the manager whenever an issue of interest is identified. To report an issue to the analyzer, state machines can use the `sm_context::warn` method, which takes as parameters:

1. A pointer to an instance of a concrete class inheriting from the `pending_diagnostic` class.
2. Either the tree (c.f., Section 2.2.1) or the svalue responsible for the issue.
3. A pointer to the GIMPLE statement triggering the issue.
4. A pointer to the node of concern in the supergraph.

A diagnostic can be seen as a list of two types of events, each pointing to a given location in the analyzed code source:

1. State change events (e.g., a pointer transitioning from allocated to freed), which represent every state change that occurs during the path's exploration.
2. Final state events (e.g., dereferencing a freed pointer). Thus, the GSA generates a report showing the line that triggers the issue, as well as all the lines that lead to this state.

4.7. GNU Attributes System and GSA Relationship

As presented earlier, GCC provides an extensive attribute system extending the C language (c.f., Section 2.3). Attributes and the GSA share a common purpose: both can be used to enhance the static analysis performed by the compiler. Therefore, it is not uncommon for programmers to run the GSA, which is off by default, on annotated code to catch more bugs or potential security issues.

An illustrative example is the `nonnull` attribute, as presented in Listing 12. The same example was used in Section 2.3, with precision regarding the component responsible for emitting the warning. Interestingly, no warning is emitted when compiled with `-Wnonnul`, even though, as stated by the GCC manual [58] this option "*warns about passing a null pointer for arguments marked as requiring a non-null value by the nonnull function attribute*". This missed emission is primarily due to the design of the warnings engine, which is built as a lightweight intraprocedural static analyzer to minimize the incurred overhead during compilation (Section 2.4). More specifically, no

Image 1: Diagnostic output example for the code presented in Listing 11, presenting a UAF vulnerability.

```c
extern void bar (int *) [[nonnull (1)]];
void foo (void) {
  int *ptr = NULL;
  bar (ptr);
}
```

Listing 12: Example of the usage of the `nonnull` attribute.

warning pass tracks the state of pointers allocated by `malloc`, nor its relatives (such as `calloc`), which limits the scope of the analysis to the most straightforward cases.

Fortunately, the GSA comes to fill this gap thanks to its malloc state machine in its interprocedural static analysis. As suggested by D. Malcolm [133], the GSA's sole developer, the initial state of a malloc-allocated pointer is *unchecked*, implying that it might be "null". Thus, an appropriate warning message is generated when the same code is compiled with the GSA option enabled (i.e., with the `-fanalyzer` option). Of course, no warning is displayed if the `nonnull` attribute was not used. This intricate interaction between the two systems allows GCC to seamlessly support new non-trivial analyses that are hinted by custom attributes and checked by the GSA.

4.8. Native GSA's warning

The GSA provides 43 new warnings to GCC's CLI, including 12 warnings tied to its internals, as presented in Table 7. The 31 remaining warnings are implemented within native state machines responsible for different kinds of analysis, as presented in Table 8. The following is an exhaustive list of the GSA's native state machine:

- **Malloc state machine:** presented earlier in Section 4.5, this state machine is responsible for detecting misuses of allocated memory objects through different APIs (e.g., `malloc`/`free`). Its behavior can be augmented to detect mismatch deallocator with custom allocation functions marked with the `malloc` attribute, as explained in Section 4.5.
- **File descriptor state machine:** implements an analysis related to the usage of file descriptors, similarly to the malloc state machine. It aims to detect misuse of file descriptors, such as double-close or incorrect access (e.g., reading a file descriptor opened in write-only mode).
- **Taint state machine:** an experimental state machine aiming at implementing a taint analysis (c.f., Section 3.2.3.2) focusing on user-input missed sanitization. To do so, it introduces a function's attribute `tainted_args`, expecting as an argument the indices of the parameters depending on user inputs.
- **Sensitive state machine:** an experimental state machine aiming at detecting when sensitive data gets leaked to unauthorized actors, linked to CWE-200 [13]. It is somewhat limited as it only relies on calls to the `getpass` function to identify sensitive data. The GSA exposes a function to potential plugins that can be called when "trust" boundary functions are invoked, detecting if a given svalue contains an uninitialized svalue (e.g., a specific field in a structure instance).

- **Signal state machine:** implements an analysis abstracting the signal handling on operating systems. Currently, it only checks if a handler is calling an unsafe function in the context of signal handling (e.g., `printf`).
- **`va_list` state machine:** this state machine detects misuses of `va_list` typed memory objects. The C standard defines this type to represent a list of arguments of varying size ([52, §7.16]).

Warning	Description	CWE-ID
`-Wanalyzer-imprecise-fp-arithmetic`	Warns when floating-point arithmetic is used in places where precise computation is needed. Limited to calls to allocating functions when a floating-point value is passed as the requested size.	*None*
`-Wanalyzer-infinite-recursion`	Warns when a function appears to lead to an infinite recursion.	CWE-647
`-Wanalyzer-jump-through-null`	Warns when a null function pointer is called.	*None*
`-Wanalyzer-out-of-bounds`	Warns when an out-of-bounds read or write operation is performed on a buffer.	CWE-119
`-Wanalyzer-putenv-of-auto-var`	Warns when a call to `putenv` is made with a pointer to an automatic or stack variable. Documentation mentions the SEI CERT C Coding Standard [134].	*None*
`-Wanalyzer-shift-count-negative`	Warns when a shift is performed with a negative count.	*None*
`-Wanalyzer-shift-count-overflow`	Warns when a shift is performed with a count greater than or equal to the precision of the operand's type.	*None*
`-Wanalyzer-stale-setjmp-buffer`	Warns when `longjmp` is called to rewind to a buffer relating to a `setjmp` call in a function that has already returned.	*None*
`-Wanalyzer-write-to-const`	Warns on attempt to write through a pointer to a `const` object.	*None*
`-Wanalyzer-write-to-string-literal`	Warns on attempt to write through a pointer to a string literal.	*None*
`-Wanalyzer-use-of-pointer-in-stale-stack-frame`	Warns when a pointer to a popped stack frame is dereferenced.	*None*
`-Wanalyzer-use-of-uninitialized-value`	Warns when an uninitialized variable is used.	CWE-457

Table 7: Exhaustive list of warnings provided by the GSA directly related to its internals.

	Warning	Description	CWE-ID
File Descriptor State Machine	`-Wanalyzer-double-fclose` `-Wanalyzer-fd-double-close`	Warn when a file descriptor is closed more than once.	CWE-1341
	`-Wanalyzer-fd-access-mode-mismatch`	Warns when a file descriptor is used without the corresponding access rights (e.g., writing in a file open in read-only).	*None*
	`-Wanalyzer-fd-leak` `-Wanalyzer-file-leak`	Warn when a file descriptor (or a `FILE *`) is not closed.	CWE-775
	`-Wanalyzer-fd-phase-mismatch`	Warns when an operation is performed on a file descriptor not in the right state (e.g., calling `accept` on a socket not passed to `listen`).	CWE-666
	`-Wanalyzer-fd-type-mismatch`	Warns when the wrong type of file descriptor is used in an operation (e.g., calling `accept` on a file and not a socket).	*None*
	`-Wanalyzer-fd-use-after-close`	Warns when a file descriptor is read or written after being closed.	*None*
	`-Wanalyzer-fd-use-without-check`	Warns when a file descriptor is used without being checked (e.g., directly after a call to `open`).	*None*
Malloc State Machine	`-Wanalyzer-double-free`	Warns when an allocated pointer is deallocated more than once.	CWE-415
	`-Wanalyzer-free-of-non-heap`	Warns when a stack-allocated object is passed to a deallocating function.	CWE-590
	`-Wanalyzer-malloc-leak`	Warns when a heap-allocated object is not freed.	CWE-401
	`-Wanalyzer-use-after-free`	Warns when a pointer is dereferenced after being freed.	CWE-416
	`-Wanalyzer-mismatching-deallocation`	Warns when an allocated object is freed with the wrong deallocator (e.g., an object allocated through a call to `malloc` and deallocated with a call to `delete`).	CWE-762
	`-Wanalyzer-null-argument` `-Wanalyzer-possible-null-argument`	Warn on a possibly or known null pointer passed to a function marked with the `nonnull` attribute (c.f., Section 4.7).	CWE-476 CWE-690
	`-Wanalyzer-null-dereference` `-Wanalyzer-possible-null-dereference`	Warns on a possibly or known null pointer that is dereferenced.	CWE-476 CWE-690

Taint State Machine	`-Wanalyzer-tainted-array-index`
	`-Wanalyzer-tainted-offset`
	`-Wanalyzer-tainted-divisor`
		Warns when a value depending on user input is used respectively as an index in an array access, an offset in a pointer arithmetic expression, or as a divisor.	CWE-129
			CWE-823
			CWE-369

	`-Wanalyzer-tainted-allocation-size` `-Wanalyzer-tainted-size`	Warns when a value dependent on user input is used as a size passed to either an allocating function or to other functions (e.g., `memset`).	CWE-789 CWE-129
	`-Wanalyzer-tainted-assertion`	Warns when a value depending on user input is used in a condition, triggering a call to a function marked with the `noreturn` attribute.	CWE-617

	Warning	Description	CWE-ID
Signal State Machine	`-Wanalyzer-unsafe-call-within-signal-handler`	Warns when a function known to be unsafe in the context of signal handling is called by a function used as a signal handler.	CWE-479
Sensitive State Machine	`-Wanalyzer-exposure-through-output-file`	Warns when a sensitive svalue is written to a file.	CWE-532
	`-Wanalyzer-exposure-through-uninit-copy`	Warns when an uninitialized value might leak data across a security boundary (e.g., a function from the kernel-space copying data to the user-space).	CWE-200
`va_list` State Machine	`-Wanalyzer-va-arg-type-mismatch`	Warns for calls to functions with variadic parameters, but the type of a passed argument does not match the expected type.	CWE-686
	`-Wanalyzer-va-list-exhausted`	Warns when calls to functions with variadic parameters are passed with a `va_list` without enough arguments.	CWE-685
	`-Wanalyzer-va-list-leak`	Warns when a call to `va_start` or `va_copy` has no corresponding call to `va_end` on the returned `va_list`.	*None*
	`-Wanalyzer-va-list-use-after-va-end`	Warns when a `va_list` is used after a call to `va_end`	*None*

Table 8: Exhaustive list of warnings related to the GSA's different native state machines.

4.9. GSA vs. the World

Previous sections focused on dissecting the GSA's internals and data structures, whereas this section will attempt to assess whether the GSA is suitable for a real-world codebase. Section 4.9.1 presents the GSA behavior over a subset of a testsuite containing tests for several CWEs, and Section 4.9.2 illustrates a CVE discovered in OpenSSL with the GSA.

4.9.1. Juliet Testsuite

The NIST Software Quality Group maintains the Software Assurance Reference Dataset (SARD) [135], which is a vast collection of test programs in various programming languages with documented weaknesses. The Juliet test suite, part of this dataset, consists of standalone C programs illustrating numerous CWEs. Among them, some tests target CWE–415 (double-free) and CWE–416 (Use-After Free)[5]. Table 10 displays the total number of tests alongside the flaws' detection rate by the GSA, while Table 9 shows the average compilation time.

The detection rate appears to be relatively low, but it actually highlights one of the GSA's current limitations. Regarding the undetected tests for CWE–415, they are not based on a single translation unit. Let's take the example of a function freeing a heap-allocated pointer and passing it to an external function, also freeing the pointer (but in another translation unit). On the other hand, undetected tests for CWE–416 highlight are based on the same construction, though the external function is dereferencing the pointer instead of freeing it. Alas, the GSA analyzes translation units individually, so it has no view on the whole program. Fortunately, as an ongoing project, discussions are underway to make the analyzer pass a subpass of the `whole-program` pass within GCC's compilation pipeline.

5. Link to the concerned tests list.

	CWE-415	CWE-416
GCC	11.3095 ms	13.0667 ms
GCC + GSA	14.7321 ms	22.4667 ms
Overhead	30 %	72 %

Table 9: Average compilation time per-file for Juliet Testsuite CWE–415 and CWE–416 for both GCC with and without GSA enabled with `-O0` optimization level.

	Number of C files	Number of detected flaws	Ratio
CWE-415	336	156	46.4 %
CWE-416	150	54	36 %
Total	486	210	43.2 %

Table 10: Detection rate by the GSA for both CWE–415 and CWE–416 Juliet testcases.

```c
sigalg = use_pc_sigalgs
? tls1_lookup_sigalg(s->s3->tmp.peer_cert_sigalgs[i])
: s->shared_sigalgs[i];
    if (sig_nid == sigalg->sigandhash) // Potential NPD
```

Listing 13: The code leading to the NPD within the `SSL_check_chain()` function.

4.9.2. Real-World vulnerability

In 2020, Bernt Edlinger, an OpenSSL developer, reported a Null Pointer Dereference (NPD) vulnerability within OpenSSL, which was discovered with the GSA's help, as mentioned in the associated OpenSSL's security advisory [136]. The CVE ID CVE-2020-1967 was assigned to this vulnerability, alongside a high score of 7.5/10 [22]. The vulnerability is exploitable by a custom OpenSSL client, triggering the NPD and a crash of the OpenSSL server, leading to a Denial Of Service [137].

In a nutshell, the vulnerability is located in the `SSL_check_chain()` function, either during or after a TLS 1.3 handshake on the server-side (c.f., Listing 13). If the client provides an invalid or an unrecognized signature algorithm, the call to `tls1_lookup_sigalg()` returns a `NULL` pointer.

As precised in Section 1.2, an artifact to reproduce our evaluation of the GSA is publicly available on Zenodo [38].

4.10. Conclusion

This chapter focused on dissecting the GSA's internals by examining the exploded supergraph construction and exploration, based on the program state and its inner region-based model, store, and constraint manager. The extensibility of state machines leverages the GSA to virtually perform any analysis that might be represented as a state machine. Together, this leaves the GSA with a high-potential compiler-based path-sensitive static analysis framework, powered by symbolic execution capabilities to trim infeasible paths.

Its evaluation highlighted both its potential and its limitations. On one hand, it is limited on a per-translation unit basis, leaving undetected vulnerabilities that human reviewers might easily spot. On the other hand, it discovered a critical vulnerability within OpenSSL, a complex and widely used cryptographic library.

Overall, the GSA remains a promising framework for researchers to implement new analyses, thanks to the abstraction of symbolic execution and the gathering of path-sensitive constraints. Its seamless integration within the GCC compiler relieves, by design, an essential part of developers' reluctance to break their workflow, whether because GCC is already part of their everyday workflow or due to the ease of integration into Continuous Integration (CI) systems.

Optimization Level	GCC	GCC + GSA	Relative
`-O3`	1.324 s ± 0.077 s	4.378 s ± 0.698 s	3.31 ± 0.56
`-O0`	433.6 ms ± 12.0 ms	3.779 s ± 0.649 s	8.71 ± 1.52

Table 11: Execution time measurements performed with `hyperfine` [138], a benchmarking tool.
The row 'Relative' corresponds to an execution time ratio compared to GCC; for example, with optimization level `-O3`, GCC was executed 3.31 times faster than the GSA.
Note: the GSA does not detect the vulnerability in `-O0` with its default configuration.

The GSA is a living, open-source project, and, like many open-source projects, it lacks developers and maintainers, even though it has a multitude of projects and ideas. It would benefit both industries and academics if researchers became involved, whether it is to improve the internals (e.g., SMT solver integration), enhance existing analyses, or implement new ones.

Chapter 5

GnuZero: A Compiler-Based Zeroization Static Detection Tool for the Masses

Coding standards for secure programming recommend "scrubbing" sensitive data once it is no longer needed; otherwise, secrets may be recovered, as illustrated in the Heartbleed attack [5]. Despite being an effective software-based countermeasure, *zeroization*, i.e., overwriting with zeroes, turns out to be challenging and error-prone. Current verification approaches suffer from scalability or precision issues when applied to production software in practice. In this chapter, we extend the GSA to build GnuZero; an automated tool that detects missing zeroization for all stack and heap variables storing sensitive data, whether directly or indirectly through inheritance. Our experiments confirm GnuZero's efficiency and effectiveness in verifying real-world benchmarks. In particular, GnuZero passes all the relevant Juliet's test programs, namely those associated with MITRE's CWE–244 and CWE–226. In addition, GnuZero succeeds in identifying new vulnerabilities in open-source cryptographic modules.

Reference: This work is the outcome of a joint work with Mohamed Sabt and Pierre-Alain Fouque. It has been published in the proceedings of the *IEEE/IFIP International Conference on Dependable Systems and Networks 2025* [24], and received one of the three *Best Papers Awards* and the *Distinguished Paper Award*.

Contents

5.1. Context and Motivations 70
5.2. Enhancing the GSA 71
5.2.1. Tracking Region's State 71
5.2.2. State Machine's API 72
5.3. Zeroization Analysis 72
5.3.1. Problem Statement 73
5.4. The Scrub Attributes 74
5.4.1. Heap Memory Zeroization 75
5.4.2. Stack Memory Zeroization 76
5.5. Zeroization with Tainting Analysis 76
5.5.1. The Tainted State 77
5.5.2. The Taint Propagation 77
5.6. Zeroization State Machine 79
5.7. Evaluation 79
5.7.1. Research Questions 79
5.7.2. RQ1: Usability and Optimization Sensitivity 80
5.7.3. RQ2: Finding Vulnerabilities 81
5.7.4. RQ3. Analyzing Complex Projects 82
5.7.5. RQ4. Complexity and Overhead 82
5.8. Related Work 83
5.9. Conclusion 84
5.9.1. Future Work 85

5.1. Context and Motivations

Zeroization of sensitive data remains a daunting issue. Indeed, secure systems must resist attackers being able to observe or provoke a system into "leaking" or revealing secret data, such as credentials and secret cryptographic keys [139]. Leaving sensitive data in memory increases the damage caused by memory disclosure vulnerabilities [140]. Non-zeroization is particularly important for any cryptosystem developed in a memory-unsafe programming language (e.g., C), or aiming to provide forward secrecy [23]. Thus, several coding standards and guidance documents call for sensitive data to be scrubbed when no longer required [141]. However, they offer little advice on how this is to be achieved or verified, especially given the complexity of programming languages and modern compilers. This lack of advice is unfortunate, primarily since the American National Institute of Standards and Technology (NIST) has defined two CWEs, aka Common Weaknesses Enumeration, that are directly linked to uncleared sensitive data: CWE–226 (stack zeroization) [7] and CWE–244 (heap zeroization)[8]. A third CWE to be considered is CWE–14 (dead store elimination) [142] that might be the cause of the CWEs mentioned above.

At first glance, scrubbing sensitive data might seem simple: just explicitly overwrite the data with zeros and proceed, right? Unfortunately, zeroization of sensitive data is far from being trivial. Z. Yang *et al.* [143] have shown that real-world programs still contain uncleared sensitive data, due to simple programmer errors, as well as poor interactions between compilers and zeroization functions. Indeed, as pointed out by C. Percival [23], "*it is basically impossible to get it right using this coding pattern*", especially that "*it would be difficult to find all the places where 'zeroization' should be inserted*".

Moreover, as highlighted by C. Percival [144] in its erratum, it is very challenging in practice to ensure that a secure zeroization function works effectively. Indeed, zeroization of sensitive data is a textbook example of DSE; most compilers recognize that sensitive data will never be read again and thus silently discard the call to scrubbing functions. This silent deletion occurs because zeroization is a non-functional property of the executed program. Incorrect or even missing implementations of zeroization do not break any functionality, thereby being more difficult to catch by automatic regression tests. Now that this problem has been identified, already with `memset()` in 2002 by M. Howard [145], multiple ad-hoc "hacks" to work around this problem have been developed; however, none of them is entirely straightforward or foolproof.

Previous work has empirically studied the most popular "secure_zeroization()" implementations in the real-world to assess their effectiveness [28,143]. They identified some successful implementations, although we argue that the scope of their conclusions is somewhat limited to a given set of compilers versions. In both [144] and [146], authors agree that it is extremely difficult to guarantee that a call to `secure\_zeroization()` will never be optimized out by increasingly smart compilers in a portable and platform-independent way. Compilers are allowed to act as aggressively as they wish during optimization, as long as they conform to the C specifications. In other words, the only reliable way for real-world projects to test whether a zeroization has occurred is to manually check the generated binary, as questioned in an issue raised by the Bitcoin Core project [147]. This manual checking is to be repeated whenever the compiler version has changed, which can be time-consuming. This difficulty underscores the need for compiler-based tools to examine code for uncleared sensitive data automatically. Furthermore, the designed tool should be friendly for developers. As a matter of fact, while designers and implementers of cryptographic software are highly aware of the problems caused by missing zeroization, awareness runs low outside this community. Thus, leakage can be unexpectedly found even inside security-related programs, since real-world projects consist of various components with unverified handling of sensitive data, as was recently illustrated by the vulnerabilities found in the PAM module [148] and the KeePass Password Manager [149] in 2023.

Contributions. In this work, we tackle this problem by designing GnuZero: a static compiler-based tool to identify CWE–226 and CWE–244 in practice. In other words, our tool automatically examines the source code for any missing zeroization of sensitive data.

Our design is based on the GSA, presented in Chapter 4. The implication is that GnuZero runs on the source level, but it operates on GIMPLE SSA, which is the architecture-independent IR of GCC (c.f., Section 2.2.2.2). GnuZero, as the GSA, runs quite late in the compilation stage, way after various optimizations, including Constant Propagation and DSE. Thus, our GSA-based tool permits a balanced compromise, as developers interact with the source code for annotation and backtrace interpretation, and the actual analysis is performed at the GIMPLE SSA level, which is the machine-independent IR closest to the actual optimized binary. Backed with this approach, GnuZero is both expressive and precise. Indeed, we do not overlook compilers optimizing away zeroization (e.g. `memset()`), while we avoid overtainting by stopping the propagation when sensitive operations are optimized to constant functions (e.g., $x - x = 0$).

To sum up our design, GnuZero is built to help developers who might: (1) forget to scrub sensitive data, (2) incorrectly implement a zeroization function that compilers shall not optimize out, or (3) pass their sensitive data to some untrusted code that may improperly handle secrets.

5.2. Enhancing the GSA

During the early stage of GnuZero's implementation, we noticed some limitations of the GSA. This section aims to discuss them and the modifications performed on the GSA. Those modifications are significant to leverage GnuZero's analysis within the GSA.

Artifacts. To modify the GSA's internals, we need to recompile GCC and modify its source code, as the GSA is included in GCC's source code. Our modifications were built on top of a fixed commit [35], and the modified source code is provided in the artifacts of this thesis.

As precised in Section 1.2, our modifications to the GSA are publicly available on the GitLab instance of INRIA: https://gitlab.inria.fr/pphilipp/gcc/-/tree/analyzer-tracking-region?ref_type=heads.

Reference: Our modification to the GSA has been presented in the *FOSDEM 2024* [36] technical conference, in the GCC developers room during the talk "*Unlocking Secret Analysis in GCC Static Analyzer*" [37].

5.2.1. Tracking Region's State

The first issue we identified was related to the state map (c.f., Section 4.5): only the state for svalues can be tracked. Let's consider a basic state machine tracking if a non-pointer typed variable marked with a `mark` attribute is used in a condition, such as illustrated in Figure 20. If the state machine tracks a variable's svalue, confusion is possible between variables, as some svalues are unique per-type and value, for example, `constant_svalue` (c.f., Figure 21). This uniqueness helps to reduce memory consumption by reducing the number of svalue instances (c.f., Section 4.3.2). To avoid this possible confusion, we modified the GSA's state map implementation to the state for either region or svalue, such as illustrated in Figure 22.

This behavior, in our opinion, illustrates the GSA's design to work with pointers, rather than memory objects. Indeed, svalues are perfect to work with pointers, as they leverage a "free" points-to analysis, but turn out to be limiting to implement analysis that needs to track l-values' state.

As previously explained, we extended the `sm_state_map` to be able to track the state of either a region or a svalue. To achieve this goal, we modified the `sm_state_map` internals by adding a hashmap from region to state, alongside the original one mapping svalue to state, and its API by adding functions to lookup and insert into this newly created region/state map.

To leverage state machines for this new state map, we modified the `sm_context` API by adding a selector parameter to the top-level functions for lookup and setting a state for a variable: `get_state`, `set_state`, and

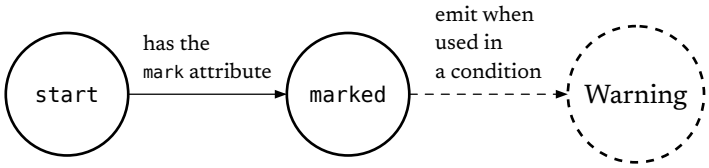


Figure 20: A dummy state machine emitting a warning when a variable with the `mark` attribute is used in a condition.

```c
1 int x = 42;
2 int y = 42;
3 char z = 42;
```

Subfigure 21.1

store	
region	svalue
decl_region(x)	constant_svalue(type: int, value: 42)
decl_region(y)	constant_svalue(type: int, value: 42)
decl_region(z)	constant_svalue(type: char, value: 42)

Subfigure 21.2

Figure 21: The code presentend in Subfigure 21.1 and its associated store upon evaluation of line 3 in Subfigure 21.2.

```c
1 int [[mark]] x = 42;
2 int y = 42;
3 if (y)
4   do_stuff ();
```

Subfigure 22.1
The program analyzed by the dummy state machine presented in Figure 20.

store

region	svalue
decl_region(x)	constant_svalue(42)
decl_region(y)	constant_svalue(42)

state_map of the dummy state machine

svalue	state
constant_svalue(42)	marked

Subfigure 22.2
Original program state's subset after evaluation of line 3 in Subfigure 22.1.

store

region	svalue
decl_region(x)	constant_svalue(42)
decl_region(y)	constant_svalue(42)

state_map of the dummy state machine

svalue's state_map	
svalue	state
empty	
region's state_map	
decl_region(x)	marked

Subfigure 22.3
Program state's subset after evaluating line 3 in Subfigure 22.1, reflecting our modifications to the GSA's `sm_state_map` implementation.

Figure 22: Subfigure 22.2 represents a subset of the program state for the program presented in Subfigure 22.1, after evaluation of line 2. It illustrates the variable's confusion, inherent to the svalue's uniqueness. When reaching statement on line 3, when the dummy state map asks for the state of `y`, the GSA will return the state of its svalue, which is the same as `x`, leading to a false-positive emission. Subfigure 22.3 represents a subset of the program state for the same program at the same program point, with our modifications on the state map reflected. Now, when the dummy state map asks for the state of `y`'s region, the GSA returns `start` as it is not tracked in the state map.

`on_transition`. These top-level functions rely on the `tree` type (c.f., Section 2.2.1), and based on the selector, our customized version will either use the region's state map or the svalue's state map.

5.2.2. State Machine's API

Once the state map was modified to track the region's state, another issue arose during interprocedural analysis. The GSA offers state machines to react when a function's frame is popped, thanks to the `state_machine`'s virtual method `on_pop_frame()`. However, state machines cannot react when a function frame is pushed, i.e., when a function with a definition in the analyzed translation unit is called, and the GSA follows the call edge from the call site's node to the callee's entry node in the supergraph. Reacting to this event is not needed for analysis tracking states for svalues, as those are seamlessly propagated through calls. When a function takes an object as a parameter, the supplied argument's svalue is copied into the parameter's associated region within the program's state store (c.f., Figure 23). We modified the `state_machine`'s API by adding the `on_push_frame()` method, and respectively modified the GSA to call it for each of its managed state machines when a function frame is pushed. This mechanism allows state machines tracking the state for a region passed as an argument to a function to reflect it to the region of the function's parameter before reaching the entry node of the called function.

5.3. Zeroization Analysis

As seen in Chapter 4, the GSA can find various problems at compile-time. When it is relevant, the GSA maps its compile-time warnings to software weaknesses, as described in the Common Weakness Enumeration (CWE) maintained by the MITRE corporation [150]. Weaknesses are mostly about security; a CWE may manifest itself as an actual software vulnerability. Unfortunately, the detection of weaknesses related to clearing sensitive data is still missing. We argue that such weaknesses are notably important for the GSA, as it was developed to scan the GNU/Linux kernel. Indeed, due to the kernel nature, uncleared data may last for a long time in the kernel memory,

```c
int g (int x) {
  return x + 1;
}

void f (void) {
  int x = 42;
  int y = f(x);
}
```

Subfigure 23.1

store

	region	svalue
frame_region(f)	decl_region(x)	constant_svalue(42)
	decl_region(y)	conjured_svalue (int y = g(x))
frame_region(g)	decl_region(x)	constant_svalue(42)

Subfigure 23.2

Figure 23: Subfigure 23.2 illustrates the store associated with the program presented in Subfigure 23.1. Each variable is associated with a region and an svalue, wrapped in its corresponding `frame_region` for clarity. In practice, the frame regions are not directly saved in the store, but referenced directly in the `decl_region`.

which might increase the impact of a kernel compromise. In this work, we focus on automatic detection for two zeroization weaknesses:

- CWE-226 (stack zeroization) [7]: sensitive information in resource not removed before reuse.
- CWE-244 (heap zeroization) [8]: improper clearing of heap memory before release.

5.3.1. Problem Statement

Takeaway: Zeroization is challenging in practice, as compilers may remove overwriting operations on sensitive data (also known as the Dead-Store Elimination problem), or worse, developers can forget to insert such operations.
To detect missing zeroization, we design a leakage model based on memory objects' lifetime, rather than on the program's termination.

Motivating Example. As indicated previously, good practice in secure programming recommends erasing sensitive data as soon as they are no longer needed. Zeroization enforces such a principle by overwriting sensitive data with zero after use. Very often, this is implemented using the code pattern in Listing 14 below.

```c
void foo (void) {
  char buf[256];
  init (buf);
  /* Some computation using buf */
  zeroize_buf (buf);
}
```

Listing 14: Zeroization example with a buffer located on the stack

The data flow of this code pattern is as follows:
1. a secret buffer is defined and allocated either on the stack or on the heap,
2. it is filled with some secret data, either by user-inputs or a random number generator,
3. some computations using this secret data are performed, potentially in place,
4. and eventually, the secret buffer's content is overwritten with 0 using the `zeroize_buf()` zeroization function.

What can go wrong in this simple example? Three things according to Z. Yang *et al.* [143]:

1. Developers might forget to call the zeroing function at each location where it is needed in the program. One might blame developers for this negligence, even though it is an understandable mistake for large codebases

and complex data flow. According to C. Percival [23], the root cause of this omission stems from the fact that programming languages lack the appropriate semantics to define sensitive variables.
2. It is often wrongly supposed that, while performing computations, no copy of sensitive data outside its original buffer is made. A copy could be made directly by a partial or complete store on another buffer (heap or stack-allocated), or indirectly through individual store or taint propagation.
3. The compiler simply removes calls to `zeroize_buf()` after optimization: an aggressively optimizing compiler could detect this final store on the buffer, without being further read, as a dead store, thus removing it and improving execution time while leaving secret data in memory.

Leakage Model. In this work, we extend the GSA to detect CWE-226 and CWE-244, namely when data identified as sensitive is not cleared. Our leakage model differs from the one defined by L.-A. Daniel *et al.* [28]. Indeed, they consider that a leakage occurs when secrets remain in the memory at the end of a program. This leakage model is quite restrictive, since it assumes that the analyzed program ends, which is not the case for many deployed programs, such as drivers and kernel modules. For instance, according to their model, uncleared sensitive data inside the kernel would leak only after shutdown. In our model, we rely on the concept of memory lifetime. The C standard specifies that every memory object (e.g., variables) exists over a portion of program execution known as its lifetime [52]. For stack variables, lifetime expires at the end of their scope (e.g., at the end of the function). As for heap objects, the lifetime ends upon deallocation, i.e., when `realloc` or `free` is called. Because accessing an object outside its lifetime is undefined behavior [52, Annex J.2], our model requires that no sensitive data remain uncleared at the end of their lifetime. To reformulate, we choose not to detect sensitive heap-allocated memories causing memory leakage, i.e., those that are never freed. Since GnuZero can detect them as leaking, the GSA also does and will emit a warning corresponding to the memory leak. Moreover, in our work, we do not consider missing zeroization because sensitive data was not identified as such. Instead, we consider 'forgotten' or 'omitted' zeroization despite being aware of the sensitivity of the data.

In what follows, we describe how we extend the C language to annotate sensitive data using GCC attributes. Then, we introduce how we design our GSA-based tool to detect the missing zeroization of sensitive data after their lifetime, whether allocated on the heap or on the stack, respectively. Our design presented in this section solves the issues discussed above, except for the indirect copy problem, which we deal with in Section 5.5.

5.4. The Scrub Attributes

> **Takeaway:** We define two attributes: `scrub` to identify variables holding secrets, and `scrubber` to identify zeroization functions. Regarding the `scrub` attribute, we use it to either taint variables' region or svalue, depending on their type: svalue for pointer-typed variables, and region for non-pointer-typed ones.
> Moreover, we also specify the bounds of a variable's lifetime depending on which memory area it is allocated (i.e., heap or stack). Stack-based memory objects live as long as their function's frame is valid, whereas heap-based ones live from a call to an allocating function (e.g., `malloc`) up to a call to a deallocating one (e.g., `free`).

Once the zeroization problem has been identified, the issue of determining which data needs zeroization remains. A quick response would be "only sensitive data is required to", but the C language has no means to identify variables as sensitive. In addition, sensitivity is context-dependent. For example, an address might be sensitive in the GNU/Linux kernel space, but not in cryptographic code. This context-dependency motivates our work to define an annotation-based C syntax for sensitive data, so that developers can annotate their own code. To this end, we rely on the GNU attribute system that is often used to extend the C semantics (c.f., Section 2.3).

In Section 2.5, we presented how user-provided plugins can augment GCC. In our case, we implement a handler for the`PLUGIN_ATTRIBUTES` event to register two new attributes:

1. *`scrub`*: a local variable attribute that identifies data as sensitive.
2. *`scrubber`*: a function attribute indicating a zeroization function, i.e., it destroys the sensitivity of its input parameter.

Regarding the source of sensitive data, we only consider local variables with no linkage, as it is not recommended to use variables whose lifetime spans the entire program execution, which includes both global variables and local variables declared with the `static` keyword. Moreover, we only consider concrete variables, excluding parameters in function declarations. Indeed, the semantics of sensitive parameters are quite different; a function should not be expected to scrub its pointer-type parameter(s) unless it is a scrubbing function, namely all functions decorated with the *`scrubber`* attribute. We consider such functions as sinks; therefore, our tool does not step down to analyze them further.

The internals of our annotation system depend on the region-based model leveraged by the GSA (c.f., Section 4.3). Indeed, at the source level, we only decorate the sensitive variable with the `scrub` attribute. However, within GnuZero, the annotated element in the state map depends on the region's type associated with the decorated variable:

1. the memory region for non-pointer typed regions,
2. the svalue for pointer-typed regions,
3. the super region for array regions, i.e., the whole array.

Note that we define the same attribute for both stack and heap-allocated variables, even though we analyze them differently. Below, we present the core of our analysis.

5.4.1. Heap Memory Zeroization

Regarding sensitive data allocated on the heap, which implies using the libc's `malloc/free` API in C, the state machine could use calls to this API as an initial transition for a variable (e.g., `malloc`) or as a sink (e.g., `free`). However, treating all memory objects as if they were sensitive would add unnecessary overhead through zeroing [139] similarly, tracking all allocated data as sensitive would generate many false positives. Hence, on the one hand, the initial transition to the *`unscrubbed`* state only occurs in two cases:

1. automatically for variables decorated with the `scrub` attribute.
2. if a variable is initialized from a variable already tracked as such.

On the other hand, the only way for a variable in the *`unscrubbed`* state to move into the *`scrubbed`* state is to be passed as an argument to a scrubbing function. Warning should be emitted by the state machine if, and only if, a variable in the *`unscrubbed`* state is freed (i.e., at the end of its lifetime).

The Special Case of Realloc. During the implementation of our state machine, the particular case of `realloc` arose. Five different outcomes can result from a call to `realloc` (without considering undefined calls); we detail them here by looking at the different C standards (from C89 to C23), alongside details of glibc 2.40 implementation and how a zeroization analysis would be impacted.

- **Failure:** since C89, `realloc` should leave its input memory object unchanged upon failure. Glibc's implementation is compliant: it returns a null pointer and sets `errno` accordingly. A warning should be emitted regarding zeroization analysis if the input memory object is not *`scrubbed`* during error handling.
- **Equivalent to `free`:** in C89, if its input memory object is not null and the requested size is zero, `realloc` is equivalent to a call to `free`. There is no precision regarding the expected behavior for C99 and C11, and since C23, such a call is considered undefined behavior. Nonetheless, the glibc's implementation kept the behavior of C89. Since such a call is equivalent to `free`, a warning should be emitted by a zeroization analysis if its input memory object is not *`scrubbed`*.
- **Equivalent to `malloc`:** since C89, if its input memory object is null, `realloc` is equivalent to a call to `malloc` with the requested size. The Glibc's implementation complies with the standard, and a zeroization analysis may want to consider such a call equivalent to `malloc` to catch all possible outcomes of `realloc`. However, our implementation does not take this case into account, because it seems unlikely to rely on this specific behavior instead of directly using `malloc`.
- **Success:** `realloc` might (1) resize its input memory object in place (i.e., its address does not change, and no new object is allocated) or (2) a new memory object is allocated, and the content of the input memory object is copied into the new object up to the size of the input object. Although the C standards do not mention the moving behavior, the terms "old" and "new" are used. The Glibc documentation specifies this moving behavior, and those two cases have different outcomes for the zeroization analysis.
  - ***Success with no move:*** in this case, the call to `realloc` has no side effect on the analysis. Undeniably, the zeroization analysis does not need any transition, as the address returned by `realloc` is the same as its input memory object's address.
  - ***Success with move:*** since C99, the old memory object has to be freed, and the address of the new memory object is returned. The Glibc implementation is compliant, and if the resizing cannot be done in place, internal documentation states a "*malloc-copy-free sequence*" is executed [151]. This specific behavior is a real issue when considering sensitive data, as there is no safe way, namely free of undefined behaviors, to scrub the sensitive data inside the old memory object upon `realloc` execution. The zeroization analysis should emit a warning specifying that this call might leak sensitive data only if it is copied into a new memory object.

Close to our findings, we highlight different programming guidelines: recommendations regarding sensitive data usage with `realloc` in the CERT C coding standard [152], the description of CWE–244 [8] for the "*malloc-copy-free*" mechanism, and in the ANSSI (the French authority for information systems' security) C guidelines [153] for the behavior equivalent to `free`. To resume, reallocating memory objects known to contain sensitive data should only be done if needed and with extra caution. As `realloc` might or might not leave *unscrubbed* sensitive data on the heap, a custom implementation of `realloc` should clean the old memory object, especially if it is sensitive.

5.4.2. Stack Memory Zeroization

When considering the nature of sensitive data in stack-based variables, the scope of our analysis focuses mainly on structures and buffers (i.e., arrays and pointers allocated through a call to `alloca`). The *scrub* attribute has no granularity over decorated memory objects of composite type, whether it is an array or structures. Even though such a behavior does make sense for sensitive arrays, structures might be composed of both sensitive and non-sensitive elements. In the worst case, this might trigger false positives if not the whole structure was scrubbed. More granularity is added for tainting variables, which is explained in Section 5.5. For our tool, there are two main differences compared to heap variables: sink and tracking. Regarding the scrubbing mechanism, we apply the same method as for heap-allocated variables, i.e., only a call to a function decorated with the *scrubber* attribute will initiate a state transition for its argument to the *scrubbed* state.

Related to the sink, recall that our leakage model expects our tool to issue a warning if there exist some variables at the *unscrubbed* state at the end of their lifetime. For stack-allocated variables, this happens at the end of their function scope. More precisely, the lifetime of variables with automatic storage duration is equal to their scope. Since C99, we are allowed to define variables at any point of the function with their own block scope (i.e., inside `{...}`). In GIMPLE, all such variables are moved to the beginning of their functions, and therefore their lifetime always expires at the end of the function. Here, we extend our state machine to examine the state of local variables whenever the GSA pops the frame of the current function during the abstract interpretation.

As for tracking, unlike heap-based variables, stack variables are not always allocated by the returned value of a function call (e.g., `malloc`). This raises the following problem: the GSA does not invoke our state machine on statements consisting of variable declarations, as they are not considered actual GIMPLE statements. Thus, the previously described state machine must be modified to track such stack-based variables correctly. Our design considers the two possible initialization methods for stack variables:

1. passed as an argument to a function by reference (e.g., its address might be passed to an initialization function).
2. through direct assignment (potentially from the returned value of a function call).

Let us discuss both.

Argument by reference: such initialization mechanisms make sense only when considering structures or stack arrays. For this case, the state machine requires a new initial transition trigger on function calls, except for specific functions (e.g., `malloc` or `memset`). For each function call, we look into the passed arguments; if they are decorated with *scrub*, we trigger their initial transition to the *unscrubbed* state.

By direct assignment: in this case, the state machine also needs a new initial transition, despite being very similar to the heap-allocated variables. The analysis must examine the left-hand side (LHS) of every single GIMPLE assignment and track its memory object as *unscrubbed* if it is decorated.

5.5. Zeroization with Tainting Analysis

Takeaway: We define a tainted state for memory objects holding or pointing to sensitive data. Although we distinguish two cases, either the memory object is tainted by:

1. attribute (i.e., `scrub`), in which case we consider it as an inextinguishable source of sensitive data, logically named *source*.
2. propagation, i.e., their assignment depends on another tainted variable (possibly a *source*), named *tainted*. *Source* memory objects can be modified in place without losing their tainted state, as opposed to *tainted* one, which can lose its taint upon taint-independent assignment.

In his thorough analysis "*zeroing buffers is insufficient*", C. Percival [23] points out that zeroization of code-source variables does not necessarily result in scrubbing every location where sensitive data might be stored. In other words, sensitive data might propagate outside their original memory space, including into memory objects that do not map to any source-level variables. The copy may either come from developers or from the compiler. Indeed,

developers may initialize some explicitly-declared local variables with sensitive data. This case applies when two or more sensitive variables are combined in some way to get a derived variable. Moreover, compilers may make copies of data into other places that we see in GIMPLE SSA as implicitly-declared and initialized local variables (c.f., Section 2.2.2.1). These variables are anonymous, since they do not appear in the source code, but in the GIMPLE SSA representation. In the best case, these anonymous variables are optimized out while lowering GIMPLE to RTL, which is the low-level GCC intermediate representation (c.f., Section 2.2.3). However, in the worst case, the final RTL code, and so is the binary, might keep the value of a sensitive variable in an implicitly allocated temporary variable on the stack, or even on a CPU register.

One might expect that the situation with sensitive data left behind in temporary variables is less problematic, since such resources are liable to be reused more quickly. Still, in fact, this can be even worse. For instance, the CVE-2023-32784 affecting KeePass was caused by some leftover strings that, once dumped, offer likely password characters for each position in the password [149]. In their work, S. A. Olmos *et al.* [139] find that data derived from sensitive data and contained in stack frames further down the call stack is not scrubbed. They give the libsodium's ChaCha20 API function as an example, where developers explicitly copy some secret key on the stack of some called functions processing that secret key. Then, they show that such residual sensitive data presents an exploitable behavior that may leak the secret key.

The many ways in which sensitive data is propagated make zeroization a daunting task. Indeed, real-world projects are made of many components that programmers did not develop. Unfortunately, it is not uncommon that such components handle sensitive data poorly, which results in leakage even in specialized security applications and libraries. Thus, traditional static and dynamic analysis techniques are limited in scope, typically focusing on a single variable, and are unable to handle more than one function call. This difficulty underscores the need for tools to aid in examining programs for missing zeroization automatically by stepping down called functions and tracking copied data.

In this section, we address sensitive data leak through explicit and implicit propagation, which requires more extensive modification to the GSA. Our design involves two central elements: first, introducing the tainted state, and second, deciding how to propagate that state over the code abstract interpretation. We examine each of these topics below.

5.5.1. The Tainted State

The term "tainting" in the GSA is traditionally referred to denote data coming from an untrusted source, as presented in Section 4.8. When the function attribute `tainted_args` is used, potential vulnerabilities are then discovered by determining whether tainted data reaches a sensitive sink [154] This definition of sensitive data differs from ours, even though the underlying mechanisms are the same.

In our model, variables are identified as sensitive either because they were defined as such or because they were derived from other sensitive variables. Thus, in addition to the *untracked* variables, we take the *`unscrubbed`* state of tracked variables and split it into two other states: *source* and *tainted*. Both states indicate sensitive data that must be transitioned to *`scrubbed`* state if no leakage shall be found. The "source" is distinguished by being decorated with the *`scrub`* attribute and only destroyed through a call to a scrubbing function (i.e., a function decorated with the *`scrubber`* attribute). Tainted variables follow the same rules as source decorated by attributes ones to determine which element (i.e., r-value or l-value) will be associated with a state, which is type-dependent: svalue for pointers, and l-value (or region) for non-pointer types.

We now need to define sensitivity inheritance, propagation, and destruction semantics for both kinds of tracked variables.

5.5.2. The Taint Propagation

Tainting occurs through code interpretation. Loosely defined, the GIMPLE language includes numerous operators that apply to sensitive data. In this work, we only consider a subset of all possible GIMPLE statements causing state propagation:

1. We limit our analysis to the C language, since it is the only language that the GSA can fully handle, eliminating any GIMPLE statement linked to other programming languages (e.g., C++ constructors).
2. Except for function calls, control-flow statements do not have any effect on state propagation, as by design, they cannot have side effects.

Thus, the operations that we consider are identified as either function calls or GIMPLE assignments, including pointer arithmetic as well as array and pointer access.

Now, let us define how variables get tainted. Below, we will first consider the intraprocedural case (i.e., from local to local), and then the interprocedural case (i.e., from the caller's local to the callee's parameter). Related to GIMPLE statements, function calls are part of the interprocedural analysis, whereas assignments are part of the intraprocedural analysis.

Intraprocedural. Given the following statement $A \rightarrow B$, we answer this simple question: what is the state of A provided B's? We say that A is the LHS variable, while B is the RHS variable. Regarding LHS, we distinguish two cases. First, if the LHS is in the *source* state, then there is no modification in its state. In other words, it does not lose its taint or transition to the `scrubbed` state regardless of the RHS's state or value. Recall that only a `scrubber` function can impact a *source* variable. Second, the LHS is not a source: it can potentially be an anonymous variable created by the compiler. Here, the LHS takes the taint of the RHS. Namely, the RHS might taint, untaint, or leave the LHS state unchanged. Unlike the source variables, tainted variables are (field/element)-sensitive, allowing our analysis to report a subset of a non-decorated array as being sensitive instead of reporting the whole array. Table 12 resumes all the different outcomes for an assignment.

The propagation policy becomes less straightforward when we consider SSA variables. Indeed, we would always have had leaks if zeroization of tainted variables were performed through a simple assignment. By definition, SSA variables are assigned in exactly one location in the program, and naturally, multiple assignments to the same variable create new versions of that variable, as discussed in Section 2.2.2.2. The policy of tainting all variables, including SSA ones, is clearly too conservative and errs on the side of tainting too much. Our design choice was to balance the desire to preserve any possibly interesting taints against the need to minimize false positives. When considering an assignment for state propagation (i.e., RHS is tainting LHS), LHS is either:

1. an anonymous SSA, then we track the state for the SSA variable itself, or
2. an SSA instance of a source-code declared variable.

In the latter, we track the actual variable declaration to allow detection of any future assignment considered for scrubbing. At the same time, if the RHS has a single operand and is an anonymous SSA variable, we choose to clear its state to avoid generating many false-positive cases.

Interprocedural. There are three cases to take into account regarding a GIMPLE call statement:

1. Is the body of the function within the compilation unit?
2. Does the function have external linkage?
3. Is the actual function behind a function pointer?

For the last case, the GSA, through different heuristics, attempts to select the best candidates for a call through a function pointer. Hence, if candidates are found, the exploded supergraph will make a direct call for each candidate, which will then fall under the other two cases, i.e., whether the body is within the compilation unit or not. Regarding the second case, i.e., if the function body is not accessible, then our analysis will either:

1. start tracking the decorated variable passed as an argument if it is seen for the first time,
2. or stop tracking any tracked argument if the function identifier matches a function decorated with the `scrubber` attribute.

The last remaining case is quite transparent regarding tracked pointers, since the GSA will propagate their state. However, the analysis has to detect pointers taking the address of the caller's decorated stack variables, as this could be the first time this address is taken (e.g., initialization functions). Non-pointer-typed variables' tracking relies on our modified version of the GSA calling our state machine's `on_push_frame` implementation (c.f., Section 5.2.2). Thus, when a variable's region is tracked as *`unscrubbed`* and passed as an argument, the corresponding parameter's region is tracked accordingly. The statements of the callee are not different from those of the caller from the state machine's point of view, thanks to the exploded supergraph. Thus, once inside the called function, we apply the intraprocedural approach, with the information relative to the call pushed to the program state.

Null pointers consideration. Each time a new condition is met, the path is split for both outcomes, i.e., `true` and `false` (c.f., Section 4.2), and the corresponding constraints are tracked (c.f., Section 4.4.2). When our analysis considers a pointer, whether for propagation, scrubbing, or emitting a warning, we ask the program state to evaluate the condition `pointer == NULL`. If the considered equality is known and `true`, then the pointer is known to be null; hence, nothing is done by our analysis. Otherwise, the analysis keeps going. This mechanism strips off the emission of false positives from our analysis.

Current state		New state
LHS	RHS	LHS
`source`	*any*	`source`
`untracked` `tainted`	`untracked`	`untracked`
	`source`	`tainted`
	`tainted`	

Table 12: State propagation through assignment.

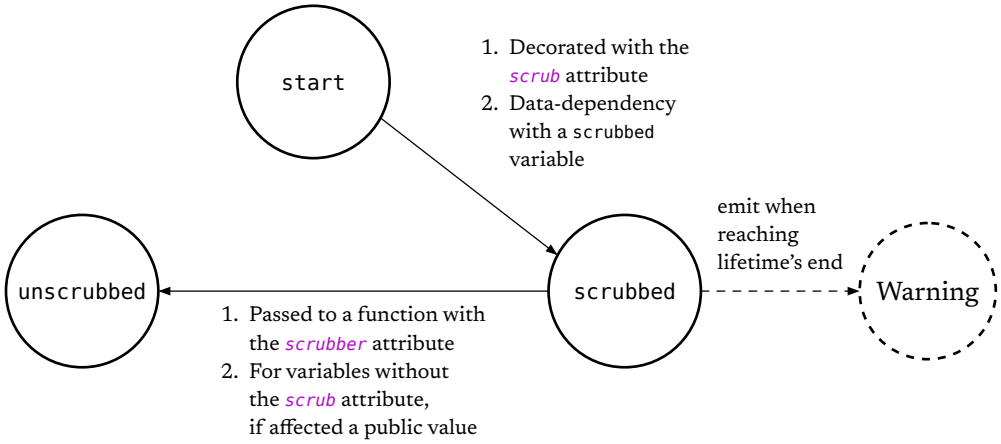


Figure 24: The zeroization state machine as implemented in GnuZero.

5.6. Zeroization State Machine

Since our design leverages the symbolic execution engine of the GSA, our tool is implemented as a state machine registered to the GSA (c.f., Section 4.5). Taking into account the different notions introduced in this section, a state machine designed to detect the non-zeroization of sensitive data would consist of a finite-state machine with two states:

1. *`unscrubbed`*: for sensitive variables waiting to be scrubbed. Used for both *source* and *tainted* variables.
2. *`scrubbed`*: for sensitive variables effectively scrubbed.

Figure 24 illustrates the state machine implemented in GnuZero. Both conditional transitions, i.e., (`start`, `scrubbed`) and (`scrubbed`, `unscrubbed`), have two different cases, illustrating the differences between *source* and *tainted* variables.

5.7. Evaluation

5.7.1. Research Questions

The purpose of GnuZero is to detect the non-zeroization of sensitive data. In order to assess its effectiveness, we investigate four main research questions:

- **RQ1. Usability and Optimization Sensitivity.** Is GnuZero easy to use and able to adapt its analysis according to the optimization level of the compiler?
- **RQ2. Finding Vulnerabilities.** How effective is GnuZero in finding vulnerabilities in programs?
- **RQ3. Analyzing Complex Projects.** Is GnuZero able to scale to analyze complex real-world projects?
- **RQ4. Complexity and Overhead.** How efficient is GnuZero when analyzing real-world code?

To answer our research questions, we conduct various experiments that we detail below.

Experiment Setup. Experiments were performed on a laptop with an 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz processor and 16GB of RAM. Unlike related work (e.g., secretgrind [118] and BINSEC/REL [155]), there is no need to start the analysis from the beginning of the main function, since we abstractly interpret the target code per compilation unit. For complex projects where the target code requires multiple dependencies from other translation units, we produce our evaluation artifact in a way that simplifies the reproduction of our experiments. Thus, the different source code files have been pre-processed manually, so that GnuZero runs on

```c
1  void heap_issue(void) {
2  unsigned size = 64;
3  char *[[scrub]] buf
4    = (char *)malloc (size);
5  if (buf)
6    {
7      char *_guard = buf;
8      buf = realloc (buf, 256);
9      // in case realloc fails
10     if (!buf)
11       buf = _guard;
12     else
13       size = 256;
14     memset (buf, 0, size);
15   }
16 free (buf);
17 }
```

Listing 16: Heap-related sensitive data usage - `realloc` issue.

```c
1  extern [[scrubber]]
2    void scrubber_fn(void *);
3  extern void init(char *);
4  extern char some_cond;
5  void foo(char *ptr) {
6    char x = *ptr;
7    char y = x;
8    y += 1;
9    y -= x;
10   use (&y);
11   x = 42;
12 }
13 void stack_issue(void) {
14   char [[scrub]] buf[256];
15   init (buf);
16   if (some_cond)
17     foo (buf);
18   scrubber_fn (buf);
19 }
```

Listing 17: Stack-related sensitive data usage.

one standalone translation unit with no further compilation dependency. For more readability, irrelevant code has been commented out from the original file.

As precised in Section 1.2, an artifact is publicly available on Zenodo to reproduce our evaluation [39].

Implementation. We implemented GnuZero on top of the GSA version 13.0.1 [35]. Hence, all analyzed programs were compiled with this version of GCC independently of the target architecture (i.e., with the flag `-c`). We run the tool using the following command line `gcc -fanalyzer -fplugin=/path/to/gnuzero.so -c`.

5.7.2. RQ1: Usability and Optimization Sensitivity

→ *Methodology.* We apply GnuZero on short programs that we craftily write to evaluate our leakage model as well as our tainting strategy. We vary the nature of the variable to scrub, heap, or stack, insert different branches and function calls to trigger different zeroization problems, then leak the same original secret through different tainted variables. In some cases, we introduce code that can be simplified or removed by the compiler optimization. For the sake of brevity, we include two representative programs of our tests (Listing 16 and Listing 17 for CWE-244 and CWE-226 respectively), and discuss the GnuZero's outputs.

→ *Findings.*

Heap Example (Listing 16). Illustrating CWE-244, this example has a weakness due to the "*malloc-copy-free*" behavior of `realloc`. We notice that the code attempts to carefully handle realloc by still scrubbing the data in case of a failure (through the use of `_guard` lines 7 and 11). However, the case in which the data are moved somewhere else on the heap might still happen. In addition to this `realloc`-based leak, a DSE-based leak is detected because of the compiler optimization discarding the call to `memset` (line 14), since the compiler identifies the source pointer as not being read until the end of its lifetime (call to `free` line 16). Such a leak is only detected when the program is compiled with optimization. Despite working on the source level, the GSA, and therefore GnuZero by design, runs after all the GIMPLE SSA-based optimization (i.e., machine-independent ones), such as DSE. Surprisingly, this simple code cannot be detected by BINSEC/REL, since it does not support dynamic memory allocation.

Stack Example (Listing 17). Illustrating CWE-226, this program does contain a weakness because sensitive data within `buf` might leak through the local variable `y` of the function `foo`. This data leak happens if `some_cond` is true, so that `foo` gets called. Sensitive data are correctly scrubbed by assignment for local `x` (line 11), and by a call to a scrubbing function for the source `buf` (line 18). The particularity of this program is that it leaks only when compiled without optimization. Indeed, the compiler detects that the final value of the local `y` (line 9) must be 1, and therefore is independent of the tainted variable `x`. This optimization, known as constant function, automatically removes

```
./heap-example.c: In function 'heap_issue':
./heap-example.c:12:13: warning: leak of unscrubbed data 'buf' [CWE-244] [-Wanalyzer-malloc-leak]
   12 |       buf = realloc (buf, 256);
      |             ^~~~~~~~~~~~~~~~~
  'heap_issue': events 1-4
    |
    |    8 |   char *TO_SCRUB buf = (char *)malloc (size);
    |      |                                ^~~~~~~~~~~~
    |      |                                |
    |      |                                (1) 'buf' is allocated here
    |    9 |   if (buf)
    |      |      ~
    |      |      |
    |      |      (2) following 'true' branch (when 'buf' is non-NULL)...
    |   10 |     {
    |   11 |       char *_guard = buf;
    |      |             ~~~~~~
    |      |             |
    |      |             (3) ...to here
    |   12 |       buf = realloc (buf, 256);
    |      |             ~~~~~~~~~~~~~~~~~
    |      |             |
    |      |             (4) 'buf' leaks here (when realloc succeed and buffer is moved); was allocated at (1)
    |
```

Image 2: GnuZero's output when run over the heap example illustrated in Listing 16.

taint when the concerned variables become constants, and illustrates the importance of systematically checking for missing zeroization each time the program is compiled.

Ease-of-use. From our experiments, we notice three main features that highlight the usability of our tool. First, running GnuZero on a given program, or more precisely a translation unit, is as simple as compiling it using GCC. Second, identifying sensitive data is as simple as adding GCC attributes to the concerned variables only once during their declaration. Third, the generated report is expressive, allowing developers to quickly understand the issue to patch it (c.f., Image 2). Every report constitutes a unique problem. In addition, reports can be viewed directly on VS Code, since they support SARIF.

5.7.3. RQ2: Finding Vulnerabilities

→ *Methodology.* Now, we would like to assess how effectively GnuZero can detect missing zeroization empirically. Here, we proceed as follows. First, we apply GnuZero on a standardized testing set of vulnerable programs. The test programs are relatively short, but they contain sensitive data in various ways that remain unclear. We compute the detection rate of GnuZero over this set, since the ground truth of these tests is known. Second, we consider GnuZero on libsodium, which is a real-world cryptographic library. In particular, we run our tool on the ChaCha20 implementation in which the authors of [139] have manually detected uncleared sensitive data down on the stack of some called function. We want to examine whether GnuZero can automatically detect such a leakage.

→ *Findings.*

Juliet Testsuite. The Juliet test suite, presented in Section 4.9.1, contains 144 tests targeting CWE–226 and CWE–244[6], split into 72 tests per CWE. These tests are the results of 18 control-flow variants of four different programs, each with weaknesses and corresponding mitigations. Given the 144 tests, we semi-automatically annotate the sensitive variables in the different source files. Here, we went through the code and noticed that the sensitive variables were consistently named `password`. We used `sed` to annotate all the code, and quickly verified that the annotation had been correctly placed. Then, we ran GnuZero, which generates all the reports in around 40 seconds. Finally, we manually reviewed all the obtained reports to compare their findings with the ground truth of these tests. Our results are promising, with a 100% true-positive detection rate and 0% false-positive or negative results. These results make GnuZero, to the best of our knowledge, the first static tool that passes all the Juliet tests related to both CWE–226 and CWE–244.

ChaCha20's API. In their work, S. A. Olmos *et al.* [139] point out that libsodium's ChaCha20 API implementation leaks some secret cryptographic key. The API is implemented in "`crypto_stream/chacha20/ref/chacha20_ref.c`", and follows a particular coding pattern. Indeed, it declares a global instance of a structure exposing static function definitions. Four different ones are exposed in this compilation unit; they differ in their IV setup, but lead to the

6. https://samate.nist.gov/SARD/test-cases/search?flaw[]=CWE-226&flaw[]=CWE-244&language[]=c&limit=25&status[]=accepted

same leaking function.
The code is as follows:

1. the secret key is passed as an argument and copied into a local context structure (only holding a sixteen `uint32_t` wide array) through an initialization function `chacha_keysetup`, copying the key into parts of the context's inner array (indexes 4 to 11),
2. this context becomes then the argument in a call to `chacha20_encrypt_bytes`,
3. and finally, all the elements of the context's inner array are used to initialize sixteen local variables that are never zeroized from the function stack.

GnuZero successfully reports that vulnerability when no optimizations are performed (`-O0`). Moreover, nothing is reported starting at optimization level `-O1`, as GCC suppresses the variables leading to the leak.

5.7.4. RQ3. Analyzing Complex Projects

→ *Methodology.* This research question involves analyzing real-world projects for missing zeroization. For this work, we limited the scope to non-trivial projects that are both complex and hard to examine dynamically or at the binary level. Here, we chose two projects handling secrets in the kernel, namely the GNU/Linux cryptographic modules and the wpa_supplicant WiFi driver. Note that state-of-the-art tools, such as BINSEC/REL, cannot analyze these two projects, because they associate leakage with program termination. For each project, we do not intend to conduct a comprehensive exploration. Instead, the purpose of our experiments is to find new missing zeroization issues within these projects. Success or failure assesses either the limitations or the effectiveness of our approach when GnuZero is applied to complex real-world projects.

→ *Findings.*

GNU/Linux Crypto Modules. We ran GnuZero on two cryptographic modules from the GNU/Linux kernel[7] [65]: Tiny Encryption Algorithm (TEA) and Data Encryption Standard (DES). Regarding **TEA** [156], and its extended version (xTEA) [157], we analyze the API functions defined in the file "`crypto/tea.c`". GnuZero detects leaks in local variables of encryption and decryption functions, with and without optimization activated (up to `-O3`). Similarly, as for **DES** and 3DES, we analyze their API functions within the GNU/Linux kernel (file "`lib/crypto/des.c`"). We focused on the functions expanding cryptographic keys for DES and 3DES (respectively, `des_key` and `dkey`), which are similar except for the return value of the DES key expansion that may be used to detect weak key usage. Our analysis successfully detects uncleared sensitive data in these two functions. An insight of our analysis is that zeroization is often forgotten when sensitive data propagates into stack frames further down the call stack, even in projects that consider zeroization seriously.

WPA Supplicant - Kernel Driver. From its manual, it is "*an implementation of the WPA Supplicant component, i.e., the part that runs in the client stations*", and "*acts as the backend component controlling the wireless connection*". The analysis was focused on the `wpa_supplicant_set_wpa_none_key`[8] [158], a function used for *ad-hoc* network. The function relies on a local buffer to copy a pre-shared key from its parameters, and a call to `memset` for zeroization is present at the end of the function. The leakage is detected when the call to `memset` is suppressed at the first optimization level (`-O1`).

5.7.5. RQ4. Complexity and Overhead

→ *Methodology.* We now want to assess how GnuZero impacts compilation time empirically. We used the `hyperfine` [138] benchmarking tool to execute GnuZero, the GSA and GCC with the same parameters for each real-world code.

→ *Findings.* Table 13 depicts the results from `hyperfine`. We can see that overhead is not much, except for DES and libsodium without optimization. We suspect the reporting system to be the main cause of this overhead. Indeed, in Section 4.6, we explained how the execution time of the reporting system is correlated with the number of paths and their complexity, leading to warning(s) reported by state machines: more complex paths induce longer execution times. The results illustrate this behavior. The best case to illustrate this is the case of libsodium: GnuZero execution time is 99% faster when optimization is turned on. The analyzed function has several local variables (`jx`) aliasing the sensitive input, which is never erased or overwritten in the source code. However, no issue is raised with the optimization version, because GCC optimizes the use of local variables by reusing them, thereby overwriting their memory space with other temporary variables. No issue implies no computation on path feasibility, which decreases execution time.

7. Commit tagged "v6.12"
8. Commit ID: f91680c15f80f0b617a0d2c369c8c1bb3dcf078b

Libsodium					
Optimization	Command	Mean [ms]	Min [ms]	Max [ms]	Relative
No (`-O0`)	GCC	57.4 ± 5.6	49.1	70.2	1.00
	GSA	2932.2 ± 241.8	2513.7	3319.8	51.10 ± 6.53
	GnuZero	3592.7 ± 147.3	3456.3	3865.4	62.62 ± 6.63
Yes (`-O2`)	GCC	28.8 ± 3.6	21.8	41.0	1.02 ± 0.16
	GSA	28.3 ± 2.6	21.3	32.9	1.0
	GnuZero	29.7 ± 2.7	22.5	37.3	1.05 ± 0.14
DES					
Optimization	Command	Mean [ms]	Min [ms]	Max [ms]	Relative
No (`-O0`)	GCC	120.8 ± 16.2	97.0	182.5	1.00
	GSA	7021.6 ± 54.9	6952.1	7099.9	58.14 ± 7.80
	GnuZero	51385.5 ± 2701.0	48652.6	56223.7	425.50 ± 61.19
Yes (`-O2`)	GCC	192.3 ± 9.1	181.5	212.3	1.0
	GSA	1082.0 ± 95.0	999.6	1329.6	5.63 ± 0.56
	GnuZero	52273.2 ± 5300.2	45709.8	59964.2	271.81 ± 30.40
TEA					
Optimization	Command	Mean [ms]	Min [ms]	Max [ms]	Relative
No (`-O0`)	GCC	44.9 ± 3.5	36.5	55.4	1.00
	GSA	73.3 ± 4.5	65.5	86.6	1.63 ± 0.16
	GnuZero	170.6 ± 6.0	159.9	179.9	3.80 ± 0.32
Yes (`-O2`)	GCC	83.0 ± 8.9	73.2	119.8	1.0
	GSA	89.0 ± 7.1	76.4	103.9	1.07 ± 0.14
	GnuZero	153.2 ± 9.6	131.5	171.4	1.85 ± 0.23
WPA Supplicant					
Optimization	Command	Mean [ms]	Min [ms]	Max [ms]	Relative
No (`-O0`)	GCC	36.3 ± 4.0	28.5	47.9	1.00
	GSA	39.9 ± 4.1	31.3	57.3	1.10 ± 0.17
	GnuZero	41.9 ± 3.0	32.9	47.9	1.15 ± 0.15
Yes (`-O2`)	GCC	30.7 ± 1.7	28.1	35.8	1.0
	GSA	32.8 ± 2.8	29.3	53.4	1.07 ± 0.11
	GnuZero	35.2 ± 3.5	31.0	55.5	1.14 ± 0.13

Table 13: Execution time measurements get through the usage of `hyperfine` [138], a benchmarking tool. We ran it on each real-world code to compare the compilation time of using GSA and GnuZero over GCC for optimization levels `-O0` and `-O2`.
The column 'Relative' corresponds to an execution time ratio to the quickest command (value 1.0).

5.8. Related Work

Already in 2005, J. Chow *et al.* [140] identified the problem of sensitive data unnecessarily remaining in memory: the longer sensitive data resides in memory, the greater its chances of being leaked by various means. Several similar discussions are found in the literature. In particular, C. Percival [23] offers some insights into the challenges of forcing a (smart) compiler to maintain zeroization of sensitive data. R. Chapman [141] follows a set of recognised guidelines and coding rules for secure programming that require the zeroisation of sensitive data. They discussed both their scope and their limitations. In what follows, we categorize existing research work into two areas: enforcing zeroization and detecting missing zeroization, which our work builds upon.

Enforcing Zeroization. The core idea is to build programs that scrub their sensitive data without manually calling the appropriate zeroization function. Researchers have developed many solutions to address this on various

levels. J. Chow *et al.* handle this on a system level by applying OS-based secure deallocation [140,159], while R. Chapman [141] defines a language-specific solution mostly in the context of the Ada and SPARK programming languages. Different research work focused on the Java programming language. Indeed, R. R. Hansen *et al.* [160] propose the first definition of secret erasure in the Java language in the context of smart cards to guarantee that secrets are unavailable after Java Card program termination in the card. A. Pridgen *et al.* [161] aim at reducing encryption keys in Java programs that run on both Oracle and OpenJDK Java Virtual Machines, J. Lee *et al.* [162] suggest changes to the Android codebase to mitigate the problem of having uncleared secret keys in Android's TLS, and J. Lee *et al.* [163] focus on solving the problem of password retention in the Android platform. Given the versatility of the Java language, these papers show that the problem of uncleared sensitive data concerns all computing devices: smart cards, smartphones, personal computers, and servers, but they are all context-dependent solutions, instead of having a universal approach. Therefore, there has been significant interest in solutions based on compilers, since they can either formally prove the flow-preservation program transformation for volatile objects as in CompCert [164], or insert stack (not heap) zeroization during program compilation (e.g., Clang [165] or Jasmin [139]). Another approach, described by S. T. Vu *et al.* [166], is to protect any code related to sensitive data from being discarded by aggressive optimization, including DSE. In practice, GCC is the only compiler that officially ensures stack zeroization; it defines the `strub` attribute, so that the annotated functions can have their stack frames zeroed out [167]. Finally, some works suggest storing sensitive data in secure locations, instead of removing data from memory. Secure locations arguably include cloud storage [168], CPU registers [169], and specialized hardware extensions like ARM TrustZone [170] as well as Intel MPK [171]. Unfortunately, such solutions rely on features that are not universally available.

Detecting Missing Zeroization. Instead of enforcing zeroisation, another line of research considers, like ours, protecting sensitive data by detecting vulnerable applications that overlook data zeroisation. Here, we do not include related work, such as the work of Y. Du *et al.* [172], that only study detecting DSE independently of the data sensitivity, even though this might accidentally lead to detecting missed zeroization. Dynamic analysis techniques, such as data-flow analysis [173], and password tracking [97], have been introduced to detect data leakage. In addition, Secretgrind [118], a tool built upon Taintgrind [120] itself based on Valgrind's MemCheck [111], is a runtime analysis tool that uses taint to track sensitive data in memory. Secretgrind propagates taint in two ways: direct assignment and arithmetic operations. Following the same direction, namely binary analysis, BINSEC/REL [28], built upon the symbolic execution engine BINSEC/SE [155] of the binary-analysis platform BINSEC [73], can be used to analyze binaries for zeroization sensitive data. The closest work to ours is BINSEC/REL, since we both rely on static analysis and symbolic execution to detect missing zeroization. However, GnuZero differs from BINSEC/REL in three main aspects. First, GnuZero works on the (C) source level, while BINSEC/REL works on the binary level. While binary tools are more precise, they cannot consider programming abstractions. For instance, BINSEC/REL can hardly track local variables for leakage. In addition, GnuZero does detect vulnerabilities related to optimizations, because the GSA interprets GIMPLE after DSE. Second, both tools diverge in their leakage model. Indeed, BINSEC/REL checks for leakage only at the program termination, thereby excluding kernels and driver modules. In contrast, GnuZero identifies leakage more locally by looking at the lifetime of variables. Thus, BINSEC/REL was mainly used to check CWE–14 automatically, namely the preservation of zeroization functions by compilers, to further the work of Z. Yang *et al.* [143] in which verification was done manually. As for GnuZero, it can be used to analyze complex programs for missing zeroization caused either by compilers or developers, i.e., CWE–226 and CWE–244. Third, GnuZero reports all state changes, thereby allowing deeper inspection of the error, for example, the source line numbers alongside the code that caused tainting. Whereas BINESEC/REL suffers from a lack of expressiveness in reports, since it operates at the binary level.

5.9. Conclusion

The effective management of sensitive data within software systems is critical for better security. This work introduced GnuZero, a compiler-based tool that detects missing zeroization in stack and heap variables containing sensitive data. Through its integration with the GCC Static Analyzer, GnuZero achieves precision and efficiency, as demonstrated in its application to real-world benchmarks and well-known cryptographic modules. Our evaluation assesses GnuZero's ability to detect vulnerabilities, including those caused by DSE, with no or few false positives. These results highlight the potential of GnuZero and the advantages of a compiler-based tool approach:

- **Automatic Detection:** GnuZero successfully identifies CWE–226 and CWE–244 instances, improving manual and ad-hoc techniques for ensuring zeroization.
- **Taint Tracking:** By leveraging the GSA's symbolic execution, GnuZero introduces tainting semantics that track sensitive data through intraprocedural and interprocedural flows, thereby detecting leakage in any stack frame.

- **Optimization-Aware:** Operating at the GIMPLE SSA level, GnuZero considers relevant compiler optimizations, such as DSE, to underline vulnerabilities introduced during compilation.
- **Scalable:** GnuZero analyzed several real-world codes from different types of open-source projects: libsodium, GNU/Linux kernel crypto modules, and wpa_supplicant.

5.9.1. Future Work

5.9.1.1. Internals

GnuZero would benefit from both new usability features and expanded capabilities to analyze more complex real-world code. Some interesting improvements on the taint propagation mechanisms:

- Field granularity to the `scrub` attribute.
- A function attribute to track its return value or parameter(s) as a source of sensitive data. Such a feature would simplify analyzing API functions without creating an execution context.
- A function attribute to declare a memcpy-like function, propagating a taint from a given argument to another.
- An analysis parameter as a propagation depth threshold, which, when hit, would stop propagation from the associated source. This parameter would especially benefit the analysis of cryptographic-related functions (e.g., encryption) because we will automatically stop tainting ciphertexts that no longer leak anything.

5.9.1.2. Extending to Different Compilers

Different GCC versions. Technically, GnuZero includes two components: the first is a GCC plugin that can be easily ported since it leverages the standard GSA API. The second component involves modifications within the GSA, which enhances portability, as it involves merging our code with the GSA of the target GCC. We have discussed GnuZero with the sole maintainer of the GSA, who seemed interested in merging our code and introducing secrets tainting.

LLVM. GnuZero relies on the GSA based on the GCC GIMPLE IR, which binds it to GCC, even though the LLVM IR and GIMPLE IR share similarities. There are three ways to port GnuZero to LLVM:

- a GSA-like module in the LLVM toolchain, which is challenging because LLVM is primarily designed for optimizations rather than program analysis. Moreover, there is no equivalent to the GNU attribute system in LLVM, even though Clang does implement a subset of the provided GNU attributes; most of them are not passed along in the LLVM IR. To the best of our knowledge, there is no easy way to add new C attributes at the source level and propagate them from Clang IR to LLVM IR.
- a plugin within the Clang SA (not LLVM), which is limited because it misses zeroization violations caused by DSE, as those are performed within the LLVM (not Clang) optimization phase.
- a GIMPLE to LLVM IR transpiler (e.g., wyrm [174]).

Since we seek a compiler-based approach, we do not consider already existing works (e.g., pitchfork [175]) built upon symbolic execution engines (e.g., haybale [176] or KLEE [84]) that perform secrets tainting on LLVM IR.

Chapter 6

Tentative Definition of the Secret Attribute in the C Language

A critical challenge in C as a general-purpose language is the absence of the notion of secret data in its abstract machine. This results in information disclosure being poorly detected by compilers that lack the required semantics to model any vulnerability related to secret leakage. Numerous dedicated tools have been defined in the literature to overcome this limitation; each comes with its own annotation rules, tainting model, and, more importantly, a narrow scope for addressing a specific disclosure vulnerability. In this work, we introduce the required C constructions to bring secrets to the GCC compiler through its system of attributes, generalizing the work introduced in the previous chapter (c.f., Chapter 5). The resulting framework, which we call GnuSecret, not only defines consistent notations and semantics to designate secrets directly in the C language but also propagates them throughout the program code by leveraging the symbolic execution engine embedded into the GSA. Of particular interest, GnuSecret is not bound to a specific vulnerability, as its modular design allows it to virtually model any vulnerability related to the MITRE's CWE–200 and its children. Our experiments confirm GnuSecret's effectiveness in analyzing real-world projects. Indeed, it succeeds not only in detecting the CWE–200 related vulnerabilities highlighted by recently published CVEs on modern cryptographic libraries, but in spotting new ones in the GNU/Linux kernel crypto module.

Reference: This work is the result of a joint work with Mohamed Sabt, Abdoulaye Katchala Mele, and Pierre-Alain Fouque. It is the subject of an ongoing submission in an academic conference.

Contents

6.1. Context and Motivations 88
6.2. The Secret Attribute 89
6.2.1. The Secret Design 89
6.2.2. Design 90
6.2.3. The Secret Function Prototype Attribute 91
6.2.4. The Secret Type Attribute 92
6.3. The Secret Propagation 93
6.3.1. The Secret Tainting Model 93
6.3.2. The Secret Propagator Functions 94
6.3.3. The Secret Destructors 94
6.4. Sinks for Information Disclosure 95
6.4.1. GnuSecret-Logger 96
6.4.2. GnuSecret-Cond 96
6.5. Evaluation 97
6.5.1. Research Questions 97
6.5.2. Experiment Setup 97
6.5.3. RQ1: Finding Existing Vulnerabilities 97
6.5.4. RQ2. Finding New Vulnerabilities in Complex Projects 98
6.5.5. RQ3. Complexity and Overhead 98
6.6. Related Work 98
6.6.1. The GCC Static Analyzer 98
6.7. Conclusion 100

6.1. Context and Motivations

Context. Many vulnerability detection techniques for C code have been proposed in recent years. While many have focused on the detection of buffer overflows and/or format string vulnerabilities, few have taken into account more subtle errors that can silently break the confidentiality of a whole system during computation. In the broad context of software development, several CWEs, especially the CWE–200 and its descendants [13], refer to the problem of leaking confidential data to an unauthorized party. *Information disclosure*, also known as data or information leakage, occurs when sensitive or confidential data is unintentionally or improperly revealed to entities that are not authorized to access it. In this work, we prefer the term "information disclosure" or "secret disclosure" instead of the term "leak", as the latter can refer to different issues depending on the context: a "memory leak" refers to poor resource management that can lead to system exhaustion. In contrast, a "secret leak" refers to the unintended disclosure of information resulting from a specific weakness.

The scope of CWE–200 is broad, as secure code with respect to CWE–200 must avoid different rules such as CWE–203 "*observable discrepancy*", CWE–209 "*generation of error message containing sensitive information* [10], and CWE–226 "*sensitive information resource not removed before reuse*" [7]; otherwise, a single violation can unobtrusively compromise the confidentiality of the sensitive data. To address this, the research community has designed various tools that can automatically detect such non-functional properties in the code. Unfortunately, custom tools were defined for each specific disclosure vulnerability. Worse, each tool proposes its own rules for secrets annotations, builds its own static analysis techniques, and implements its own reporting system [25–29]. Such a discrepancy has created confusion for the concerned developers who are mostly unwilling to support multiple external tools [30], primarily when they address one problem at a time.

This work starts with the following observation: despite their differences, all vulnerabilities implying information disclosure share a common feature – each can be cast as a problem of information-flow violation [177] – sensitive information from secret "source" may flow into a leaking "sink" without being properly cleaned by a "destructor". This view obviously bears resemblance to the vulnerabilities related to calling sensitive computation with unsanitized user-controlled inputs, which are also seen as an information-flow violation [178]. Based on this abstraction, K. Ashcraft *et al.* [179] present a new extendable compiler-based tool to automatically detect all such vulnerabilities caused by unsanitized users' inputs.

Contribution. Inspired by their approach, in this work, we design a unified framework to detect CWE–200 and siblings instead of implementing a distinct analysis for each sink. Unlike existing solutions, we do not leverage custom tools that are unlikely to enjoy broad adoption [28,175,180] instead, we embed our framework into the GCC compiler, and more specifically, into the GSA (c.f., Chapter 4). Our design is based on the fact that many of the abstract properties relevant to information disclosure differ only in the concrete sink. For instance, to check CWE–209, we shall ensure that no secret is involved when calling log-printing functions. Thus, we can check for CWE–209 while using the same source annotation and secret propagation defined for similar related vulnerabilities.

Of course, to detect a vulnerability, the tool must first know it. The problem is that vulnerabilities concerning secret disclosure are often domain or even system-specific. Therefore, hard-wiring a fixed set into a given tool is ineffective. The core of our design is an extended generalization of the GCC attributes defined for GnuZero (c.f., Section 5.4), by proposing a new GCC `secret` attribute that embeds all semantics a developer needs to track secrets in C code. This `secret` attribute introduces the required language-level secret-memory semantics, as was strongly advocated in [181]. Our resulting framework, henceforth referred to as *GnuSecret*, defines a unified notation and semantics for designating secrets in C programs.

A final practical result about our design is that GnuSecret, as we will show in Section 6.4 for two distinct CWE entries, can be extended with little incremental development cost to include other information disclosure vulnerabilities. Our tool allows programmers or security researchers to mainly focus on modelling such vulnerabilities by clearly mapping them to concrete code actions, while leaving the bulk of the analysis to our framework. Thus, GnuSecret is intended to become a unifying framework, where different disclosure, or leakage, models can be defined to perform comparative analysis without designing yet another ad hoc secrets annotation system for this purpose.

This chapter is structured as follows. Section 6.2 describes the semantics of the different attributes that we define to designate secrets. Section 6.3 sketches our tainting model, including our constructions to declare C functions to conditionally propagate or unconditionally destroy the secret taint. Section 6.4 shows how GnuSecret can be extended to independently detect CWE–203 "*Observable Discrepancy*" and CWE–209 "*Error Message*

Containing Sensitive Information", with an evaluation on real-world projects in Section 6.5. Section 6.6 compares this work to other work in related areas, and Section 6.7 concludes this work.

6.2. The Secret Attribute

Takeaway: We propose a set of several attributes aiming at identifying secret variables. The intended goal of the different attributes is to mimic the various ways a secret can be declared: by function call, by design, or simply by type inheritance.

Modern compilers (e.g., GCC and Clang) can detect typical programming mistakes, such as Out-Of-Bounds (OOB) errors and integer overflows or underflows, at compile-time through warnings. Compiler-issued warnings serve as an early aid for developers to avoid subtle, but potentially serious, mistakes that could otherwise have severe consequences. Both criticized and appreciated, warnings have proven helpful in increasing the quality of code among developers [182]. Therefore, there is a recent trend to extend the warnings engine with its somewhat limited intraprocedure static analysis by a more complex module supporting a context-sensitive static analysis via symbolic execution. This tendency allowed compilers to detect more profound weaknesses more precisely. For instance, the GSA in GCC 15 includes 26 entries of CWEs.

However, by analyzing the supported CWEs, we notice that the GSA only offers poor support to CWE–200 [13] and siblings that concern the disclosure of sensitive information (c.f., Section 4.8). We argue that this constitutes a severe limitation to the GSA, since it implies that an important class of vulnerabilities still goes undetected. Many programs handle sensitive information, such as passwords or encryption keys, and must ensure that this data is not exposed to unauthorized parties. It turns out that this property is not straightforward to model by a compiler, mainly because it occurs in various ways. The ad hoc approach considered by the GSA for CWE–200 is not suitable to capture other weaknesses related to secret disclosure. This issue is mainly due to compilers primarily attempting to faithfully model the semantics of C as specified by the ISO standard [183]. Additionally, non-functional properties, such as power consumption or secret leakage, are not specified by the C standard. Of particular interest, since the GSA acts on the abstract C machine as defined by GCC, it fails whenever it needs to model the notion of "secret".

In this work, we fill the gap and introduce secrets into the C language through GCC attributes. The GSA can, therefore, identify all variables holding sensitive data, which constitutes the primary prerequisite to spot any related disclosure. Below, we describe our approach and the rationale behind our design choices.

6.2.1. The Secret Design

The term abstract machine refers to the conceptual model used to define how a programming language operates. In the case of C, the abstract machine's state includes, among others, variable values, the call stack, and the actual statements being executed. However, as stressed previously, this abstract machine omits several essential aspects related to security, such as the ability to recognize secrets. In their work, V. D'Silva *et al.* [184] suggest that code annotations are one way to help preserve security guarantees in source code. In this work, we consider the GCC attributes system to implement such annotations.

Types vs. Attributes. Another design could have been a new C keyword designating a type specifier for secret variables as presented by [185]. We did not consider this alternative design for two reasons. First, attributes are better suited for contextual keywords that can be ignored without altering the program semantics. Second, unlike other type specifiers (e.g., `const`), secret variables mutate their state [186] –i.e., variables might begin their lifetime as public, then they become secret during the program execution, before eventually losing their secret state as a result of data erasure. This approach does not fit for a strongly typed language like C, which resulted in many cases of false-positives [185]. In addition, the GSA is a read-only pass, and therefore cannot freely modify the assigned or inferred types in the built Abstract Syntax Tree (AST). Though similar, our attribute-based static approach differs from the type inference one.

Secret Sources. Recall that the GSA manages a set of state machines to track the different states of a program alongside their transition events. For our purpose, we build a custom state machine in which variables can either be secret or public. Naturally, all variables, except the annotated ones, are set to public by default. We design the secret attribute to be parsed and interpreted by the GSA to update the related state machine accordingly. The main challenge of this approach is to delimit the scope of the secret attribute accurately. Indeed, the GCC documentation defines four types of attributes: functions, variables, types, and statements (c.f., Section 2.3). A naive implementation would have been to restrict the secret attribute to defined variables, since they are the only

```c
/* only the private_key field of all instances of this struct is secret */
struct RSA_keys {
  char public_key[256];
  char private_key[256];
};

/* output_key becomes secret when the function returns */
int generate_secret_key (char *output_key);

/* passwd is a local variable and becomes secret on its definition */
int my_function (void) {
  char *passwd = get_user_input ();
  validate (passwd);
}
```

Listing 18: Different semantics of creating secret data.

language artifacts that are modelled as memory objects. Nevertheless, this design does not reflect all the different developers' practices for designating secrets in a program.

We illustrate the different constructions that a programmer can use to define a secret source in Listing 18, as secrets can take various forms in code. The most typical form involves local variables that store sensitive information, such as user passwords. However, secrets are not limited to variables alone – they can also be defined by the flow a program executes. For example, a function prototype might indicate that its return value or specific pointer-typed parameters become secret upon execution. Furthermore, developers might specify that all variables of a specific type become secret. These additional syntactic constructions illustrate some of the challenges faced by any design of the secret attribute, which must include the necessary C idioms to ease developers' workload when annotating secrets.

Secret Plugin. The compilation process of GCC can be extended through user-defined plugins, similarly to LLVM. These plugins are then loaded during runtime, provided that they implement the required interface from the GCC plugin API. A GCC plugin can introduce new C elements that it parses and interprets during compilation. Based on this system, we define **GnuSecret** as a GCC plugin that performs three main roles:

1. it introduces all the needed attributes to label secret data,
2. it instantiates a new GSA state map to keep track of the sensitive components in the program,
3. and it augments the GSA symbolic execution with a "taint analysis"; i.e., creates new secrets from existing ones.

Below, we detail the different secret attributes, and defer the two others to Section 6.3.

6.2.2. Design

Naively, an attribute for marking variables as secret seems sufficient, but several cases arise, primarily due to the C dialect itself and the GIMPLE IR analyzed by the GSA, as previously stated. Hence, several attributes are needed to match common programming cases. We will focus on several aspects of C dialects here: objects, functions, and types.

6.2.2.1. The Secret Variable Attribute

Takeaway: The `secret` attribute is designed to decorate variables or function parameters. It is inspired by the `scrub` attribute presented in GnuZero, meaning that we also use the type of the memory object to choose whether to taint its region or its svalue.

Our system of secrets tracking is based on the region-based model used by the GSA (c.f., Section 4.3). Strictly speaking, variables become secret once their associated memory region gets assigned sensitive data. At the source code level, we define the `secret` attribute to mark sensitive variables. Internally, we add the right memory element in the secret state map depending on the type of the annotated variable. Indeed, similarly to GnuZero, we track the memory region for primitive variable types (e.g., `int`), the svalue (i.e., the address) for pointer-typed variables, and the super memory region for both arrays and structures. GnuZero was lacking field granularity on secret declarations. The `secret` attribute defined by GnuSecret can track an individual struct field on a variable instantiating a

```c
struct foo {
  int a;
  int b;
}
void f (void) {
  // Only s.a will be considered as secret
  struct foo [[secret ("a")]] s = {0, 0};
}
```

Listing 19: Simple C code snippet illustrating the field granularity of the `secret` attribute, thanks to its optional parameter.

structure. For instance, a structure of two elements can be secret either entirely, or only its first or second field, thanks to this new attribute.

Syntactically, our `secret` attribute is defined with an optional argument to specify one or several fields to mark as secret. The argument is of string type containing the name of the desired fields, such as illustrated in Listing 19.

Semantically, a secret memory region does not taint another memory region just because they share the same svalue (c.f., Section 5.2.1). In contrast, two pointers having the same svalue must have the same secret state. Here, we only consider the pointer indirection as secret, but we track the svalue to benefit from the points-to analysis of this memory model [83] (c.f., Section 4.4.1). Structures follow similar rules:

- **Non-pointer-typed field:** the *secret* state is located in the region of the specific field.
- **Pointer-typed field:** it has the same tainting model as a pointer to secrets, meaning its svalue gets tainted.
- **Pointer to non-pointer-typed field:** the *secret* state is located in the region of the expression `ptr->field`, following the same logic applied to non-pointer-typed variables.
- **Pointer to pointer-typed field:** the *secret* state is located in the svalue of the expression `ptr->field`, following the same logic applied to pointer-typed variables.

In our implementation, it is essential to note that we distinguish the case of variable declarations from function parameters, even though we keep the same attribute name at the source level.

Variable Declarations. While iterating on each GIMPLE statement, we insert the appropriate entry into our state map whenever we encounter a GIMPLE statement that concerns a variable decorated with the `secret` attribute. It is because the GIMPLE SSA representation separates declaration from initialization, which implies that such a naive implementation unfortunately does not cover the case in which local variables are declared but initialized through a function call. We manage this case by iterating over the function call arguments and tainting the one marked by the `secret` attribute if it has not already been tainted.

Function Parameters. We can also use the `secret` attribute on parameters of functions, excluding prototypes without bodies. The parameter of the called function becomes secret at the beginning of the function execution. More precisely, the state of the concerned parameters is switched to secret when they are pushed on the stack by the function `on_push_frame` within the GSA.
In case the secret parameter is of a non-pointer type, there is no impact on the corresponding argument in the calling function, since the parameter only copies the argument value but not its memory region. In contrast, a pointer to a secret parameter results in a secret argument at the point of execution that directly follows the function call (c.f. Listing 20). This is because we track the pointer's svalue, rather than the memory region that holds the svalue directly. The key benefit of annotating function parameters is to ease developers' workload. Instead of having to mark every local variable passed to a given function, it is simpler to mark the desired parameter(s) as always being secret(s). This attribute fits the need for some cryptographic functions in which secret keys are part of their parameter list.

6.2.3. The Secret Function Prototype Attribute

> 🔑 **Takeaway:** The semantics of the `secret` attribute on function parameters means that the memory object is tainted upon entry of the function. This function attribute eases the burden of annotation, since it can be used directly on function parameters known to manipulate sensitive data (e.g., encryption function).

In C, a function prototype is a statement that informs the compiler about the function's name, its return type, and the number of parameters, along with their types. In GNU C, and more recently in C23, we can use function

```c
void bar (int * [[secret]] ptr) {
  // Upon entry of this function
  // ptr will be considered as secret
}
void foo (void) {
  int x = 0;
  bar (&x);
  // x is now secret
}
```

Listing 20: C code snippet using the `secret` attribute.

attributes to specify specific function properties, either globally (e.g., `noreturn`) or regarding some of its parameters referred to by their positions within the function parameter list using some attribute arguments (e.g., `nonnull(1)`) (c.f., Section 2.3). Function attributes serve two purposes:

1. they substitute or augment the compiler analysis of the called function, especially when the definition is not provided,
2. and they indicate some meta-property that could not have been deduced otherwise.

In GnuSecret, we define a new attribute, named `secret_out`, that can be used on all function prototypes, including for functions without bodies within the analyzed translation unit, in headers, for example. Unlike other GCC function attributes, our `secret_out` can directly annotate parameters, rather than requiring positional attribute arguments. This design choice, motivated by better code readability, raises the following challenge: only type attributes are considered for function prototypes. The problem is that type attributes do not correspond to our approach of labelling memory regions on our state map. To address this, we constantly check for the calling context and mark the related argument (not the parameter) as secret, which implies that `secret_out` can only be used on parameters of type pointer. Regarding semantics, as its name indicates, the associated calling argument becomes secret at the end of the function execution. Implementation-wise, we distinguish two cases.

1. When the function is defined, the state of the related argument is switched to secret when the GSA calls the code for leaving the function context (i.e., `on_pop_frame`).
2. When the function is not defined, the argument becomes secret at the call site by the function `on_call` in the GSA.

A final note on the syntax, the position of `secret_out` (and `secret`) does not matter as long as the attribute is placed on the function parameters or the function type. In addition, we can annotate more than one parameter of the same function.

The primary interest of `secret_out` is that it captures the notion of secret beyond the traditional view of sensitive data. Indeed, secrets are not limited to some users' data, but also include data created through a given execution flow. For example, we know that key derivation functions are meant to derive secret keys; therefore, instead of manually annotating all output buffers passed to such functions, we can use `secret_out` to annotate the relevant function once and leave GnuSecret automatically track the new secret variables. Similar to the `secret` attribute, optional arguments can be provided to specify individual fields for parameters of structure type or a pointer to a structure type.

6.2.4. The Secret Type Attribute

Takeaway: The `secret_t` type attribute allows developers to highlight that every memory object of this type has to be considered secret. This attribute eases the burden of annotation for developers, as only one annotation is needed at the type definition, rather than on each instance of this type.

In addition to marking individual variables and function parameters, we introduce the `secret` type attribute, which allows developers to annotate types as secret. This attribute helps simplify the declaration of sensitive data within structures, as it ensures that any variable declared with this type is automatically treated as secret. This attribute can be applied to `typedef` declarations and to individual fields in structures. When applied, all instances of the type will inherit the secret property, making it easier to manage sensitive data without the need to annotate each variable at declaration. Consider the following example:

```c
struct key_pair {
  int [[secret_t]] private_key;
  int public_key;
};
```

In this case, any variable of type `struct key_pair` will have its `private_key` field treated as secret, while `public_key` remains non-secret. The key advantage of using the secret type attribute is to avoid repetitive annotations. For instance, in security applications where structures contain sensitive data, such as a username and a password, or public and private cryptographic keys, developers can mark only the sensitive fields once in the type declaration. From then on, all instances of the type will automatically be treated as secret without further annotation effort. For the sake of coherence, the secret type attribute behaves identically to the secret variable one: the associated memory region, svalue, or super memory region is marked as secret in the internal state map on assignment, and the same propagation and tainting rules apply.

6.3. The Secret Propagation

So far, we have concentrated on the secret annotation problem: designating secrets in C code by using a set of attributes decorating variables, functions, and types. Obviously, we do not expect programmers to annotate all secrets, especially since adding and maintaining annotations on every secret in the program would be prohibitively time-consuming [185]. Typically, in the same manner than GnuZero did, we distinguish source secrets from tainted ones: any variable that is assigned to a value derived from other secret data will be itself marked as secret, thereby requiring to track the propagation of secret data along variable dependencies through each of the program operations [178]. This approach is well-established and supported in multiple languages, including Perl since 1994 for sanitizing user inputs [187] and Zig for constant-time operations [188]. Since the GCC 14 release in 2024, the GSA includes a form of taint analysis for users' input sanitization: the GSA defines six warnings for execution paths where tainted (i.e., unsanitized) data is used [189], as illustrated in Section 4.8. Unlike the Perl or Zig runtime tainting, the GSA applies static taint analysis during program compilation. In this work, we extend the GSA tainting engine to include our defined secret attributes. Below, we introduce our propagation model, which involves statements, undefined functions, and destructors that declassify secrets.

6.3.1. The Secret Tainting Model

> 🔑 **Takeaway:** The taint model is similar to Gnuzero's one. Meaning that the *source* and *tainted* memory objects do not follow the same propagation rules for taint destruction.

Recall that GnuSecret uses the GSA's region-based memory model to track secrets in memory through their regions and svalues elements. When a non-pointer type becomes secret, we taint its region (l-value) with the secret state. As for pointers, we taint their svalue (r-value) with the secret state, so that the pointer itself is not secret, but the memory it points to is secret. Whenever we taint an svalue of an expression or a variable, we make sure that the region of the pointed memory is also tainted with the secret state. This way, we ensure that any access to the memory location will be correctly tracked. Beyond variables at the source level, we also taint temporary variables that GCC creates for the SSA form. Unlike GnuZero, anonymous SSA temporary variables are not untracked upon assignment to named SSA variables (c.f., Section 5.5.2). Similar to secret source, tainted variables are also field-sensitive, allowing finer-grained tracking of secrets within arrays or structures.

GnuSecret relies on the GSA's path-sensitive exploration of the exploded supergraph, as presented in Section 4.2. Unlike the work of S. Wang *et al.* [25], that sets public variables to concrete values, both public and secret variables in the GSA are symbolized when required. Thus, a variable in the same code block can be both secret and public, depending on the path taken, according to the program's symbolic states for precise tainting. While symbolically going through the code and its execution flow without having to inline functions (unlike the work of [26]), when an assignment occurs, the variable, aka LHS, may get or ignore the taint of the RHS.

We preserved the same distinction as in GnuZero: we distinguish source and tainted secrets, although both represent sensitive data that shall not be disclosed. Recall the difference in semantics: source variables are explicitly annotated and can only be cleared by destructors, while tainted variables become secret through path exploration. For both C-level and temporary variables, GnuSecret enforces the following rule: source variables remain unchanged unless destructors are applied, such as scrubbing functions in GnuZero. In contrast, tainted variables always change state based on the RHS. Thus, GnuSecret has the same advantage as GnuZero: developers

are just required to annotate variables on their declaration, which is mostly at the beginning of the analyzed function, instead of looking for the buried final initialization in the code.

The special case of struct. Regarding structures, a secret super region transfers its secret state to its individual elements, but not vice versa, meaning that tainting a struct of two fields is not equivalent to individually tainting its two elements. This results in the following tainting rule: individualized or inherited tainting is mutually exclusive; for example, a specific field cannot be independently untracked if the whole structure was previously tainted. Thus, GnuSecret provides both coarse-grained and fine-grained taint tracking for structure elements, unlike what was proposed by GnuZero or by K. Ashcraft *et al.* [179], where only inherited states are implemented.

6.3.2. The Secret Propagator Functions

Takeaway: The `propagator` function attribute enables developers to mark functions that are known to propagate secrets from a given parameter to either another parameter or the return value. It expects two arguments, the indices of both the source and destination parameters. GnuSecret will then propagate the taint of the source to the destination, if, and only if, it is tainted.

External libraries are often used in real-world projects to provide common tasks, such as data processing, networking, or mathematical computations, which can be error-prone to implement independently. Despite being beneficial for software development, external libraries pose new challenges for taint analysis, as they constitute a blind spot for secret propagation. Indeed, static analysis techniques stop their exploration when they encounter a call to a function without a definition. This halt in the analysis implies that they cannot deduce how their secret arguments would taint the return value or other pointer arguments. For instance, in `memcpy(dst, src)`, without an additional rule, the argument `dst` would not be tainted even if `src` is secret. In an attempt to address this problem, existing tools include a hard-coded simplified implementation for standard libc functions, such as `memcpy` [24,28]. This solution does not scale, as it does not allow developers to express their own tainting rules for arbitrary functions.

For this purpose, we introduce the `propagator` attribute. This attribute is designed for use on a function that propagates secrets from one parameter to another parameter or the return value. It takes two arguments: a source and a destination. Conceptually, the source parameter taints the destination one; in other words, the destination becomes secret if, and only if, the source is secret. Thus, it becomes possible to skip analyzing complex functions; it suffices to apply their tainting rules. Similar to `secret_out`, the exact taint moment depends on whether the function is defined. If yes, the destination parameter becomes secret at the end of the function execution. Otherwise, the destination becomes secret at the function call site.

Regarding syntax, the propagator attribute is placed on the function type or the function definition. It can be repeated more than once on the same function to denote multiple propagators. Its two arguments are strings indicating the position of the parameter in the function signature. To specify the return value, the index must either be negative or “ret” or “return”. Note that only the destination can denote the return value of the function. As with the secret attribute, an optional field name can be additionally specified for structure types or pointers to structure types, both in the source and destination parameters. The complete syntax becomes `index[.field]`, where `index` is either a number or “ret” or “return”, and the optional `.field` is the name of the concerned structure element when relevant. Two examples are given in Listing 21.

6.3.3. The Secret Destructors

Takeaway: The `destructor` function attribute follows the same logic as the `scrubber` attribute for GnuZero: it identifies functions that destroy the taint of their pointer-typed parameters.

Following our tainting model, secrets remain secret even after they are erased. In addition, ciphertexts are also secret and must never be disclosed, since they were computed using some cryptographic secret key. This over-

```c
// Propagates from "src" to "dst.field".
[[propagator("0", "1.field")]] void my_func(int *src, struct my_struct *dst);

// Propagates from "src_2.field" to the return value.
[[propagator("1.field", "ret")]] int my_func(int *src_1, struct my_struct *src_2);
```

Listing 21: Examples of propagator functions.

tainting must be appropriately addressed because it surely results in numerous erroneous warnings that developers must sort through and painfully ignore. This is where secret destructors come into play. Indeed, destructors provide an escape hatch from strict information flow tracking. The secret `destructor` attribute is defined to indicate functions that are responsible for destroying secret data. Destroying does not necessarily mean scrubbing; encryption functions, for instance, behave as destructors with respect to ciphertexts. The ability to destroy, aka declassify, secrets provides the opportunity to release information based on some contextual analysis, as examined by M. Abadi [190] for security protocols. Supporting destructors ensures that secrets are not only tracked and propagated but also properly destroyed when no longer needed.

Destructors do not need to have a body, as the secret state removal can be triggered at the point of the function call. Regarding implementation, we iterate over the arguments and remove the secret state from the memory regions, svalues, or super memory regions depending on their type. The consequence of this is that, for non-pointer types, the associated memory region in the calling context remains secret, since it is unaffected by the call. Therefore, secret destructors operate only on pointer-type arguments. Regarding semantics, after calling a secret destructor function, all pointer arguments previously marked as secret lose their secret status. Disclosing them does not generate warnings anymore, as they are no longer considered secret.

6.4. Sinks for Information Disclosure

> **Takeaway:** We designed GnuSecret as a secret propagation framework that allows for the easy integration of sinks for various information disclosure weaknesses. Or in other words, we abstracted the sinks from the propagation. We implemented two sinks targeting different CWEs: GnuSecret-Logger for CWE-209 and GnuSecret-Cond for CWE-203.

The purpose of GnuSecret is not to merely identify and propagate secrets, but to spot CWE–200 in programs during compilation.

GnuSecret aims to spot CWE–200 in programs at compile-time, and not only to identify and propagate secrets. This is the role of sinks that are modelled to look for secrets under specific program actions. The diversity of CWE–200 implies multiple sinks to be defined. Mostly, tools are specialized, as they address one sink at a time. The proliferation of such specialized tools unexpectedly made developers reluctant to adopt them due to their custom architectures and incompatible functionality, resulting in a steep learning curve [30]. In this work, we take a novel approach: our goal is to provide all the necessary components for security experts to facilitate the implementation of their own vulnerability detector for any targeted domain or application. Our design is depicted in Figure 25, where three connected yet separate elements appear: the attributes system, the propagator and the sinks manager. These elements interact with the state map to insert, delete, and query secret variables.

Of particular interest, the `sink_manager` class handles a set of sinks that are iterated over after taint propagation at each statement. Several examples of adding a new sink exist in GnuSecret's source code. In a nutshell, it is required to:

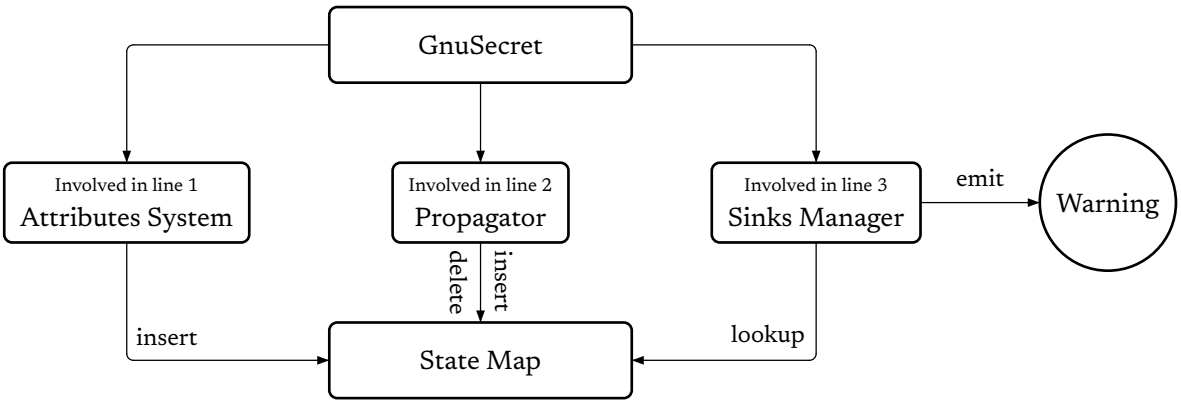
1. Define a concrete class based on our abstract class `sink` to implement a `verify` method called by the sink manager.
2. Add a sink constructor to the sinks manager.
3. Add the new source file of the sink to the building script.

To demonstrate our modular approach, we implement three sinks. Of course, GnuSecret can be extended to include more use cases. The first implemented sink is debug-oriented: we introduce the function `__is_tainted` that emits a warning when its argument is tainted (c.f. Subfigure 24.1). Alongside this sink, we implemented two other sinks, focusing on distinct CWEs, both children of CWE–200: GnuSecret-Logger for CWE–209 and GnuSecret-Cond for CWE–203.

It is worth noting that GnuSecret does not measure the amount of leaked data by its sinks. Indeed, CWE–200 related vulnerabilities vary in their potential impact: some may result in the immediate disclosure of the entire secret data, others may only reveal a single bit, and some may not be exploitable at all. However, from a security engineering perspective, such vulnerabilities should be absent, especially since novel attack techniques could successfully extract secrets.

```c
int SECRET x = get_random ();
int y = x;
__is_tainted (y);
```

Subfigure 25.1
Example code using `__is_tainted`



Subfigure 25.2
GnuSecret Components Architecture.

Figure 25: GnuSecret Modular Design.

6.4.1. GnuSecret-Logger

6.4.1.1. Design

CWE-209, formally named "*Generation of Error Message Containing Sensitive Information*", focuses on sensitive data leaking through two different program mechanisms: logging or error emission [10]. Usually, these mechanisms are implemented in functions. Some basic logging functions and error-related functions are already internally recognized by the sink (e.g., `printf`, `error`, or `write`). Still, in the same way as for previously described destructors or propagators, programs might rely on custom functions to log or handle errors, hence the need for a new function attribute: the `logger` attribute. GnuSecret-Logger detects such patterns in programs and relies on the secret propagation introduced in Section 6.3 and the `logger` function attribute. It emits a warning when tainted data gets passed to a function identified as a sink disclosing its secret arguments.

6.4.1.2. Usage

Besides the secret annotations of GnuSecret, this sink also needs to know which functions should be considered as leaking the secret. Listing 23 illustrates a snippet of C code picturing an information disclosure through a custom logging function.

6.4.2. GnuSecret-Cond

6.4.2.1. Design

CWE-203, named "*Observable Discrepancy*", denotes discrepancies as an abstract notion that can have "*many forms*" and its "*variations may be detectable in timing, control flow, communications[...], or general behavior*", potentially leading to sensitive data leaks [11]. This weakness encompasses a weak form of Constant-Time (CT)

```c
extern [[logger]] my_log(const char *, ...);
void foo(void) {
  int [[secret]] x = 42;
  // Warning is emitted here
  my_log ("%u", x);
}
```

Listing 23: C Snippet for GnuSecret-Logger, corresponding to CWE-209.

```c
void foo (void) {
  int [[secret]] x = 42;
  int y = x + 5;
  if (y) // Warning is emitted here
    do_stuff ();
}
```

Listing 24: C Snippet for GnuSecret-Cond, corresponding to CWE-203.

vulnerabilities, specifically when a program branches over a secret. Accordingly, GnuSecret-Cond only aims to detect whenever a condition is secret-dependent, leaving the program vulnerable to side-channel timing or micro-architectural attacks, and relies on the secret propagation implemented in GnuSecret. Still, conditions do not always appear in the form `lhs op rhs`, but can also be deferred to functions such as `memcmp`, which are not safe. The sink recognizes some standard comparison functions (e.g., `memcmp`, `wmemcmp`, or `strcmp`). Yet, programs can rely on custom functions that are known to have observable discrepancies regarding their arguments. This mechanism introduces the need for an attribute to identify such functions: the `nsaf` (*unsafe*) function attribute.

6.4.2.2. Usage

Similarly to GnuSecret-Logger, its input is the same as GnuSecret, i.e., annotations using the different `secret` attributes and, if needed, annotations on leaking functions. Listing 24 shows a small snippet of C code presenting a CWE–203 violation, since a secret-dependent variable is used in a condition.

6.5. Evaluation

6.5.1. Research Questions

The purpose of GnuSecret is to perform secret propagation and allow sinks registered within its sink manager to detect issues accordingly. To evaluate its effectiveness, we investigate the following three research questions:

- **RQ1. Finding Existing Vulnerabilities.** How effective is GnuSecret in identifying known CVEs in software?
- **RQ2. Finding New Vulnerabilities in Complex Projects.** Is GnuSecret able to detect new CVEs in a huge codebase?
- **RQ3. Complexity and Overhead.** How efficient is GnuSecret when analyzing real-world code?

To address these questions, we conduct a series of experiments, which we describe in detail below.

6.5.2. Experiment Setup

Experiments were conducted on a laptop with an 11th Gen Intel(R) Core(TM) i7-1165G7 @ 2.80GHz processor and 16GB of RAM. In contrast to prior work (e.g., BINSEC/REL [28]), our approach does not require initiating the analysis from an exposed entrypoint (e.g., the `main` function or an exported one). Instead, GnuSecret abstractly interprets the target code on a compilation unit basis. For complex projects involving multiple interdependent translation units, we prepared our evaluation artifacts to facilitate reproducibility. Specifically, source files were manually pre-processed such that GnuSecret could analyze a single, self-contained translation unit without requiring additional compilation dependencies. To enhance clarity, non-essential code was commented out in the original files.

As precised in Section 1.2, artifacts are publicly available on Zenodo to reproduce our evaluation [40].

Implementation. GnuSecret was implemented atop the GSA version 13.0.1 [35]. Accordingly, all target programs were compiled using this GCC version, using the `-c` compilation flag. The tool is executed via the following command: `gcc -fanalyzer -fplugin=/path/to/gnusecret.so -c`.

6.5.3. RQ1: Finding Existing Vulnerabilities

→ *Methodology.* We now aim to empirically evaluate the effectiveness of GnuSecret in identifying known CVEs. To assess GnuSecret on real-world targets, we used CWE–203 as a filter on the NIST CVEs database, published over the last three years, and identified three candidates: CVE-2023-5981 [191], CVE-2024-25189 [192], and CVE-2023-5992 [193]. Respectively, those CVEs correspond to `GnuTLS` [31], `libJWT` [33], and `OpenSC` [32], and each is linked to its RSA implementation. Our goal is to determine whether GnuSecret can detect known vulnerabilities, thereby demonstrating its ability to scrutinize complex real-world codebases.

→ *Findings.*

GnuTLS. It is a broadly used library implementing the TLS protocol. A vulnerability was found in its key exchange mechanism based on RSA, which led to CVE-2023-5981 [191]. We ran GnuSecret on the source code just before the patching commit[9]. The target is `_gnutls_proc_rsa_psk_client_kx`, which is called during the TLS key exchange if RSA PSK mode is enabled in the build. This function uses the return value of `gnutls_privkey_decrypt_data` to branch into different parts of its code, resulting in a padding oracle. GnuSecret successfully reports the vulnerability, without any false positives.

9. Commit ID: f2fbef2c50952270eeeadebfacbf718da845fadc

LibJWT. It is a JSON Web Token (JWT) library written in C, and it also implements the JSON Web Signature (JWS) RFC [194]. CVE-2024-25189, which was fixed upon its disclosure, is due to the use of non-constant-time comparison during signature verification [192]. GnuSecret analyzed the source code of the last commit before the fix[10]. The vulnerable function is `jwt_verify_sha_hmac` and was successfully reported as vulnerable by GnuSecret, without any false positives. Nevertheless, we had to comment out some complex function that makes the GSA abandon its exploration. Indeed, the GSA steps over a function when it does not have its definition.

OpenSC. It is a C library designed to communicate with smart cards. Targeted by CVE-2023-5992 [193], the vulnerability is a variant of the Bleichenbacher attack [195]. We ran GnuSecret on the last commit before the fix[11]. GnuSecret successfully detects the vulnerability, alongside a single false positive due to the missing feature of array granularity at secret declaration.

6.5.4. RQ2. Finding New Vulnerabilities in Complex Projects

➔ *Methodology.* This research question involves analyzing real-world projects for information disclosure-related vulnerabilities. For this work, we limited the scope to the GNU/Linux kernel, specifically to its RSA implementation, to be consistent with the previously studied CVEs. This project is both complex and challenging to examine dynamically or at the binary level. Success or failure assesses either the limitations or the effectiveness of our approach when GnuSecret is applied to complex real-world projects.

➔ *Findings.* The GNU/Linux kernel implements several cryptographic primitives for general purposes, both symmetric and asymmetric ones. We fixed the analyzed commit to a released version of the kernel[12]. Within the file `crypto/rsa-pkcs1pad.c`, the RSA padding code is located. We ran GnuSecret analysis on `pkcs1pad_decrypt_complete`, and a vulnerability, similar to the one in OpenSC, was discovered with a single false positive, also linked to the missing feature of array granularity at secret declaration. The information disclosure concerns branching on the two-byte header of the PKCS#1 v1.5 encryption padding, leading to a Bleichenbacher attack.

6.5.5. RQ3. Complexity and Overhead

➔ *Methodology.* We now aim to assess the impact of GnuSecret on compilation time empirically. There is no need to study the runtime overhead, since GnuSecret is a static analysis tool without any dynamic instrumentation of the resulting binary.

The benchmarking tool `hyperfine` [138] was used to execute GnuSecret, the GSA and GCC with the same parameters for each real-world code and measure the execution time.

➔ *Findings.* The results from `hyperfine` lead to a mean overhead factor of 81 % compared to GCC. None of the cases here were specifically abnormal ($<$ 500ms), though we know that the GSA reporting system can highly increase the compilation overhead, as explained in Section 4.6.

6.6. Related Work

6.6.1. The GCC Static Analyzer

Roughly, GnuSecret defines a new state machine in the GSA and introduces a family of secret attributes to interact with developers. Although uncommon, a similar design was proposed in some related work, as we will describe below.

The file descriptor state machine [196]: The GSA can emit a warning whenever a file has not been closed or is used without the proper rights. Since GCC 13, the GSA introduces a new state machine to cover other vulnerabilities for misusing file descriptors. For instance, a developer might attempt to write into a file that was previously opened in read-only mode. Similar to GnuSecret, the new state machine defines a set of corresponding attributes to feed its initial states: `fd_arg_read` for read access, `fd_arg_write` for write access, and `fd_arg` for both.

GnuZero (c.f., Chapter 5): The foundational work to this work, as it leverages the GSA to address the problem of CWE–226, which is related to information disclosure. There are three main limitations compared to GnuSecret.

1. The `scrub` attribute has coarse granularity, as structure fields cannot be tainted individually. This over-tainting might result in more false positives.

10. Commit tag: v1.15.3
11. Commit ID: f39c9d2da5170ac60de8ffd105040c9694dab11a
12. Commit tag: v6.15

2. GnuZero only allows variables to be decorated. In contrast, our family of secret attributes can also annotate types and functions, resulting in increased expressivity in our tainting model. Developers can thus define a better tracking policy following the logic of the analyzed program. In addition, GnuZero has no way to indicate arbitrary propagator functions, such as `strcpy`, `strcat`, `memmove`, or even `memset`.
3. The tainting model of GnuZero is tailored to the zeroization problem, as it defines specific tainting rules for the SSA variables to avoid related false positives. This tainting model is not adapted for a framework destined to universally address CWE–200 like ours. Moreover, the sinks are hard-coded, unlike GnuSecret.

Taint Analysis. Taint analysis [90] is a foundational technique in program analysis, employed to trace the propagation of sensitive data. It applies to a wide set of security applications, including the detection of exploit [197], privacy leakage [198], and cryptographic key misuse [199]. Over the past several decades, numerous taint analysis frameworks have been developed to examine vulnerabilities across diverse contexts. Notably, many of these systems incorporate hard-coded, application-specific logic to identify particular classes of errors [28,93,200–202]. While such approaches are effective for detecting predefined issues, they lack flexibility, which results in the proliferation of tools with a narrow scope. In contrast, this work advocates for a general and extensible framework capable of accommodating a broader spectrum of application-specific secrets. Furthermore, existing language-based solutions often mandate extensive and invasive code modifications [203]. GnuSecret, by comparison, is built upon GCC without adding incompatible syntactic constructs to the C language. Thus, it offers a balanced compromise embedded into a mainstream compiler to detect a broad scope of information disclosure vulnerabilities in any C program.

Constant-Time Analysis. Constant-time (CT) programming is a widely adopted paradigm for mitigating micro-architectural timing attacks, ensuring that program behavior remains independent of secret data. However, implementing CT code poses significant challenges. Over the past fifteen years, various tools have been developed to detect CT violations using techniques such as abstract interpretation [27,180,204], symbolic execution [25,26,28,205], over-tainting path exploration [29], and binary instrumentation [206]. While CT encompasses a broader scope, we focus here on tools analyzing legacy C code, excluding approaches based on domain-specific languages [203,207]}.

Unfortunately, empirical analyses continue to reveal persistent limitations of existing tools [30,208,209]. This is because many of the published approaches have focused much attention on satisfying properties about completeness or soundness, thereby ignoring tainting expressivity both on the source level and on the generated report. In numerous tools [28,29,205]}, only global variables or function parameters can be annotated as secret. In addition, to designate secrets, some tools require writing additional code, either by calling some special functions or through third-party programs [205]. Such overhead incurs a high maintenance cost that most developers are unwilling to pay [30]. In GnuSecret, we carefully design our set of secret attributes to meet the needs of the software development process. For instance, we include recommendations of A. Geimer *et al.* [208] to support local variables, since secrets can also be generated within the analyzed functions. We define types and functions attributes to cover all relevant cases without annotating every single occurrence of the concerned variables. As for propagation functions, they are implemented differently when they are supported: BINSEC/REL suggests stubs to replace system calls [28], where HAYBALE/PITCHFORK proposes function hooks to be specified in Rust [205]. In our architecture, we strive to support all desired functionalities coherently via the GCC attribute system, which developers are familiar with. GnuSecret is not only distinguished by its rich tainting syntax compared to CT tools. Indeed, our tainting model involves different tainting rules for source and tainted secrets. Our model is motivated by the fact that secrets are not read-only, since developers might continue modifying secrets in place while keeping their semantics. Thus, in GnuSecret, developers are just required to annotate variables on their declaration, mostly at the beginning of the function, and not on their final initialization.

A final remark, GnuSecret does not check that a given program is constant-time. Despite similarity with CT, our support for CWE–203 does not include all CT rules; in particular, we do not check for operator-dependent instructions, such as division or memory access. It would be an interesting future work to implement a CT sink for GnuSecret and to conduct an extensive comparative analysis with other CT tools. Relying on GIMPLE, the resulting tool would omit vulnerabilities introduced by the GCC architecture-dependent optimization [210,211]. We think that the best approach would be to rely on a compiler-based tool like ours while coding to spot source-level issues quickly, and to combine it with binary-oriented tools, like BINSEC/REL, to assess the absence of vulnerabilities with stronger guarantees.

6.7. Conclusion

This work has involved several new contributions: GnuSecret extends the C programming language and supports many language features that have not been previously integrated with static information disclosure, including local variables, individual struct fields, and an expressive syntax to propagate secrets for external undefined functions conditionally. GnuSecret also supports programmable *declassification*, allowing for secret modelling that better fits the semantics of a given program without over-tainting. GnuSecret is a compiler-based tool: our syntactic constructions are defined upon the GCC attributes, and our static taint propagation leverages the symbolic execution engine of the GSA. Finally, we show how GnuSecret can be easily extended to practically spot information disclosure, particularly CWE–203 and CWE–209.

While the approach works well on real-world code, there are three main caveats to keep in mind. First, GnuSecret requires manual annotation of secret sources. Manual annotation can get tedious in big projects, and more importantly, information disclosure can be overlooked in case of omitted annotation. Existing techniques of automatic annotation [212,213] are rather specialized for a given class of secrets (e.g., cryptographic keys), which contrasts with GnuSecret's goal to be general-purpose, as the C language. Second, we do not support fine-grained granularity when annotating arrays, which can lead to false positives in some of our evaluations. Implementing this feature would constitute an interesting improvement to our tainting model. Finally, based on the GSA, the scope of GnuSecret is limited to GIMPLE and its related optimization transformations, such as DSE. Consequently, it misses vulnerabilities introduced by architecture-dependent optimization, such as specialized hardware instructions [210]. The GSA is an evolving project, and hopefully it will be extended to RTL, the low-level GCC IR. As already announced in Chapter 4, there is ongoing work to support C++ for GCC 16, which will expand the scope of GnuSecret.

Part III

Conclusion and Future Work

Chapter 7

Conclusion

Software security remains a complex problem, as vulnerabilities rooted in memory management and secret handling continue to affect widely deployed systems, as illustrated throughout this manuscript. The initial observation of this thesis is that weaknesses such as missing zeroization and unintended information disclosure persist despite decades of research and engineering best practices. By embedding vulnerability detection directly into the compiler, we sought to bridge the gap between state-of-the-art program analysis and tools already part of developers' workflow.

Our different contributions demonstrate that compiler-based static analysis has a high potential to improve software security without burdening developers with new tooling. Through a detailed study of the GCC Static Analyzer (GSA) and by extending it with new analysis, we showed how secret-related vulnerabilities can be detected automatically and integrated into existing toolchains. This conclusion revisits the main contributions of the thesis, highlights the lessons learned from their design and evaluation, and outlines opportunities for future work.

Contents

7.1. Summary of Contributions 104
7.2. Limitations 104
7.3. Future Works 105
7.4. The Big Picture 105
7.5. Discussion: *to Warn, or to Lint, that is the Question* 106

7.1. Summary of Contributions

This thesis was guided by three research questions introduced in Chapter 1. We revisit them here and summarize how each was addressed.

I. *How suitable is the GSA as a foundation for security-related analyses?*

To answer this, we provided a detailed dissection of the GSA's internal in Chapter 4. We documented how its exploded supergraph, combined with its region-based memory model, is used to represent programs. We also dived into its state machine, the core data structure responsible for advanced analysis, and its reporting system, using information from the program state to lazily determine path feasibility and decide if a given warning should be emitted or not. We evaluated the GSA in Section 4.9, both on a public testsuite and by reproducing a CVE detected by its analysis at first hand. This work clarified both the strengths and the limitations of the GSA, and helped identify possible improvements through which new security-oriented analyses could be implemented. This dissection served as the foundation for the subsequent contribution of this thesis.

II. *Can an analysis, focusing on an information leakage use-case, be implemented within the GSA?*

We addressed this through the scope of non-zeroization of sensitive data, a specific kind of information leakage, resulting in the design and implementation of GnuZero, as presented in Chapter 5. By extending the GNU attribute system in Section 5.4 and designing a taint model based on variables' lifetime rather than the program's termination in Section 5.5, GnuZero was able to detect when sensitive data remained uncleared in memory. Our evaluation on the Juliet Testsuite in Section 5.7 demonstrated perfect precision for CWE-226 and CWE-244. At the same time, real-world analyses uncovered zeroization issues in security-critical software such as libsodium, the GNU/Linux kernel, and wpa_supplicant. The case of the wpa_supplicant driver is interesting, specifically because it highlights both the DSE problem and the advantage of a taint model based on a variable's lifetime rather than program termination. These results confirm that an analysis targeting information leakage is implementable within the GSA, while providing practical support for detecting the absence of sensitive data zeroization.

III. *How can a GSA-based analysis be generalized to detect broader information leakage weaknesses?*

Based on the result obtained with GnuZero, we sought to generalize the information leaks detection to a broader set than CWE-226 and CWE-244. We explored this through the development of GnuSecret, as presented in Chapter 6. By generalizing a set of attributes in Section 6.2 and combining a propagation model and abstract sinks in Section 6.3, GnuSecret generalizes the zeroization problem into a framework for a broader set of information disclosure weaknesses, i.e., CWE-200 and its descendants. Our evaluation showed that GnuSecret was able to detect known CVEs in major cryptographic libraries and uncovered a previously unknown vulnerability in the GNU/Linux kernel's Crypto API. This work demonstrates the feasibility of secret tracking mechanisms within the GSA and highlights its potential for addressing information disclosure directly into a broadly used compiler.

Together, these contributions demonstrate that the GSA is a promising static analysis framework for implementing security-related analysis, ranging from narrowly scoped vulnerabilities, such as missing zeroization, to more general cases of information leakage. Moreover, since the GSA is an ongoing project, Gnuzero and GnuSecret will benefit from the GSA's evolution. Overall, this thesis attempts to move vulnerability detection closer to everyday developer workflows, while highlighting the compiler's role as an essential actor for software security.

7.2. Limitations

While this thesis demonstrated the feasibility and effectiveness of compiler-based analyses for secret-related vulnerabilities, several limitations must be acknowledged.

1. The approach depends on developer-provided annotations. Although the set of attributes from the different contributions enables precise tracking of sensitive data, they rely on developers correctly identifying all relevant variables and functions. Indeed, developers could forget to insert an annotation on a variable, or worse, not be aware that this variable is holding sensitive data. Hence, missing annotations will most certainly lead to false negatives, limiting the analysis.

2. The scope of the analyses is restricted to what can be expressed in GCC's GIMPLE IR. While the IR offers a suitable abstraction to represent programs in a language-independent manner, it is also machine-independent. As a result, the tools resulting from this thesis cannot detect vulnerabilities that could be introduced by specific architecture-dependent optimizations, such as timing side-channel vulnerabilities, for example.

3. The analyses were designed and evaluated primarily for C code compiled with GCC. Although the principles are transferable to other languages and compiler frameworks, it would require significant engineering effort to replicate these tools on top of LLVM.
4. Regarding performance and scalability, the tools were able to analyze large codebases such as the GNU/Linux kernel and wpa_supplicant. However, analysis overhead may still be significant in CI with constraints on execution time.

These limitations do not undermine the contributions of this thesis. Still, they provide essential context for interpreting the results and motivate some of the future work discussed in the next section.

7.3. Future Works

The limitations identified above and highlighted within the different contributions suggest that future research is needed. Reducing those limitations would help extend the impact of this thesis and broaden the applicability of security-related analyses embedded in the GCC compiler.

1. Our contributions rely on a modified version of a specific GCC commit [35]. To benefit from the latest features and internal changes of the GSA, a rebase of our custom GCC version is required. Moreover, we aim to integrate our modifications into the upstream compiler, which would simplify the use of our tools. We presented our modification in a technical conference [37], receiving some interest from the GCC developer community.
2. Integrating the non-zeroization analysis from GnuZero within GnuSecret would be interesting. The main challenge here regards the SSA variables, as presented in Section 5.5.2.
3. Implementing an analysis to detect Constant-Time (CT) programming violations would significantly strengthen our approach. Such work would address classes of attacks that remain outside of GnuSecret's scope so far. Indeed, GnuZero is only able to detect when secret-dependent data are used within a condition instruction. To fully detect CT violation would require sinks for secret-dependent memory access and instructions known not to be constant time (e.g., division).
4. GnuSecret is a research tool resulting from this thesis. It is of particular interest to industrialize it, to perpetuate its development, and make it a stable static analysis tool of the ecosystem.

These future works underline that the contributions of this thesis are not endpoints, but rather foundations for further research.

7.4. The Big Picture

Looking at the big picture, this thesis humbly tries to reconcile the software security and software engineering communities. Indeed, a lot of work is carried out by the research community, leading to a long list of tools targeting specific vulnerabilities or weaknesses [25–29]. Such abundance leads to confusion among the developers [30]. Moreover, as demonstrated by several studies and feedback, developers are still on the lookout for a static analysis solution that does not break their workflows [18–20,128,129]. As compilers are already part of the software development workflow and embed static analysis to leverage their optimization passes, they are good candidates for new security-related analysis.

All of this put together motivated the contributions of this thesis: to explore how compiler-based analyses can contribute to improving software security. At first, by focusing on the problem of zeroization, and then tackling the broader information disclosure issue, it showed that they can be addressed directly within GCC itself. The dissection of the GSA provided the technical foundation, while GnuZero and GnuSecret demonstrated practical tools that uncover vulnerabilities in real-world software. Overall, even though limitations remain, our results confirm that security-related static analysis integrated in GCC is a promising solution to improve the security of software systems.

7.5. Discussion: *to Warn, or to Lint, that is the Question*

It is possible to argue about what is a legitimate warning and what is not. Compiler's introduced issues are a well-known side-effect of compiler's optimizations [165]. But what about issues fixed by the compiler's optimizations? In other words, if the source code is vulnerable but the compiler's optimizations mitigate or suppress that vulnerability, should an emitted warning be considered as a false positive or not? And, respectively, in the absence of a warning emission in such a case, is it considered a false negative or not?

An answer to that question is not obvious, as it depends on what is considered: UB at the source language level or during the program execution. A person focusing on compiler-independent analysis could argue that a compiler fixing an issue does not preserve the program's semantics, just as the introduction of vulnerabilities does not. Hence, labelling the missed warning emission as a false negative. However, someone focusing on the vulnerability of their compiled program could argue the opposite: no vulnerability is present in the binary, so no warning emission is a true negative. And reciprocally, if a warning had been emitted, it would have been a false positive. Those are opinions influenced by the background and objectives of the community to which one feels close.

The remainder of this section reflects my personal thinking on the subject.

I believe some discussions are needed to resolve the possible conflict emerging from this, involving both the security community and the software engineering community. Indeed, what is a "warning" is not consensual, as previously discussed. In my personal opinion, we need to fix a precise definition of what a "warning" is. During my thesis, I had discussions with several people from different communities, inspiring some leads. The concept of "lint" already exists for highlighting issues in a source language (e.g., a syntax violation such as a missing ";" for a C statement). Hence, a first sketch of a definition, where *lint* and *warning* could be used to express different ideas. *Lints* could be attached to the source language's semantics and syntax, meaning that if a given piece of code violates one of them, a *lint* should be emitted to inform the developer about it, independently of the behavior's preservation by a compiler. Whereas *warnings* could be attached to the program's behavior *during* its execution, including behavior introduced during compilation. Of course, such definitions do not describe disjoint sets of issues and only make sense for compiled languages, such as C. However, they do not target the same "view" of the program: lints are focusing on whether or not a program is *idiomatic* for the source language, whereas warnings are focusing on whether or not a program is *safe* regarding its execution.

Let's take back the example from Section 2.4. The C code presented in Subfigure 25.1 is non-idiomatic, since it presents an UB, here an Out-Of-Bounds (OOB). As explained earlier, GCC has a rather aggressive optimization pass for some standard functions, such as `strlen`, and is not able to detect the issue in the C code. This missed detection is due to the pass responsible for detecting OOB, intervening after the optimization pass deleting the call to `strlen`. With the definitions previously introduced, there is no confusion about whether the non-emission is a false negative or not. It is a false negative *lint*, not a false negative *warning*.

Of course, this definition is neither a standard nor a formal definition. It is rather a cobblestone added to the path towards one, and only reflecting my personal opinion on the matter.

```c
1 i = 17;
2 return strlen ("hello" + i);
```

C

Subfigure 26.1

```
1  i = 17;
2  _1 = (sizetype) i;
3  if (_1 <= 5) goto <D.3147>; else goto
   <D.3148>;
4  <D.3147>:
5  _2 = (sizetype) i;
6  D.3146 = 5 - _2;
7  goto <D.3149>;
8  <D.3148>:
9  D.3146 = 0;
10 <D.3149>:
11 return D.3146;
```

High GIMPLE

Subfigure 26.2

Figure 26: Example from Section 2.4, introducing a compiler's fix.

Bibliography

[1] Lucas Hobe, “Ciblée par une cyberattaque, la clinique de la Sagesse à Rennes est l'énième victime d'une longue série.” [Online]. Available: https://france3-regions.franceinfo.fr/bretagne/ille-et-vilaine/rennes/ciblee-par-une-cyberattaque-la-clinique-de-la-sagesse-a-rennes-est-l-enieme-victime-d-une-longue-serie-3042256.html

[2] FranceInfo and French Press Agency (AFP), “France Travail victime d'une nouvelle cyberattaque, les données de 340 000 demandeurs d'emploi potentiellement concernés.” [Online]. Available: https://www.franceinfo.fr/internet/securite-sur-internet/cyberattaques/france-travail-victime-d-une-nouvelle-cyberattaque-les-donnees-de-340-000-demandeurs-d-emploi-potentiellement-concernes_7394245.html

[3] Andi Yu, “Major Australian port operator shuts down amid cyber security incident, impacting goods in and out of the country.” [Online]. Available: https://www.abc.net.au/news/2023-11-11/dp-world-australian-ports-cyber-security-incident/103094358

[4] Prossimo Internet Security Research Group, “What is memory safety and why does it matter?.”

[5] “Heartbleed Website.” [Online]. Available: https://heartbleed.com/

[6] Multiple Contributors, “SEI CERT C Coding Standard - MSC18-C. Be careful while handling sensitive data, such as passwords, in program code.” [Online]. Available: https://wiki.sei.cmu.edu/confluence/x/stYxBQ

[7] The MITRE corporation, “CWE-226: Sensitive Information in Resource Not Removed Before Reuse.” [Online]. Available: https://cwe.mitre.org/data/definitions/226.html

[8] The MITRE corporation, “CWE-244: Improper Clearing of Heap Memory Before Release ('Heap Inspection').” [Online]. Available: https://cwe.mitre.org/data/definitions/244.html

[9] National Vulnerability Database, “CVE-2025-1759 Details.” [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2025-1759

[10] The MITRE corporation, “CWE-209: Generation of Error Message Containing Sensitive Information.” [Online]. Available: https://cwe.mitre.org/data/definitions/209.html

[11] The MITRE corporation, “CWE-203: Observable Discrepancy.” [Online]. Available: https://cwe.mitre.org/data/definitions/203.html

[12] National Vulnerability Database, “CVE-2025-49128 Details.” [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2025-49128

[13] The MITRE corporation, “CWE-200: Exposure of Sensitive Information to an Unauthorized Actor.” [Online]. Available: https://cwe.mitre.org/data/definitions/200.html

[14] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. in Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[15] JetBrains, “The State of Developer Ecosystem 2023 - C language.” [Online]. Available: https://www.jetbrains.com/lp/devecosystem-2023/c/

[16] Free Software Foundation, “GCC 10 Changes.” [Online]. Available: https://gcc.gnu.org/gcc-10/changes.html

[17] GSA Webpage, [Online]. Available: https://gcc.gnu.org/wiki/StaticAnalyzer

[18] B. Johnson, Y. Song, E. R. Murphy-Hill, and R. W. Bowdidge, “Why don't software developers use static analysis tools to find bugs?,” in *ICSE*, IEEE Computer Society, 2013, pp. 672–681.

[19] M. Christakis and C. Bird, “What developers want and need from program analysis: an empirical study,” in *ASE*, ACM, 2016, pp. 332–343.

[20] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, “Lessons from building static analysis tools at Google,” *Commun. ACM*, vol. 61, no. 4, pp. 58–66, 2018.

[21] NSA Center for Assured Software, “Juliet C/C++ 1.3.” [Online]. Available: https://samate.nist.gov/SARD/test-suites/112

[22] National Vulnerability Database, “CVE-2020-1967 Details.” [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2020-1967

[23] C. Percival, “Zeroing buffers is insufficient.” [Online]. Available: https://www.daemonology.net/blog/2014-09-06-zeroing-buffers-is-insufficient.html

[24] P. Philippe, M. Sabt, and P.-A. Fouque, “GnuZero: A Compiler-Based Zeroization Static Detection Tool for the Masses,” in *Dependable Systems and Networks*, IEEE, 2025.

[25] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, “CacheD: Identifying Cache-Based Timing Channels in Production Software,” in *USENIX Security Symposium*, USENIX Association, 2017, pp. 235–252.

[26] R. Brotzman, S. Liu, D. Zhang, G. Tan, and M. T. Kandemir, “CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation,” in *IEEE Symposium on Security and Privacy*, IEEE, 2019, pp. 505–521.

[27] S. Blazy, D. Pichardie, and A. Trieu, “Verifying constant-time implementations by abstract interpretation,” *J. Comput. Secur.*, vol. 27, no. 1, pp. 137–163, 2019.

[28] L.-A. Daniel, S. Bardin, and T. Rezk, “Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure,” *ACM Trans. Priv. Secur.*, vol. 26, no. 2, pp. 1–42, 2023.

[29] Z. Zhang and G. Barthe, “CT-LLVM: Automatic Large-Scale Constant-Time Analysis,” *IACR Cryptol. ePrint Arch.*, p. 338, 2025.

[30] J. Jancar *et al.*, “"They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks,” in *SP*, IEEE, 2022, pp. 632–649.

[31] GnuTLS developers, “GnuTLS Website.” [Online]. Available: https://www.gnutls.org/

[32] OpenSC developers, “OpenSC Website.” [Online]. Available: https://github.com/OpenSC/OpenSC/wiki

[33] LibJWT developers, “LibJWT Website.” [Online]. Available: https://libjwt.io/index.html

[34] “Secrets in Compilers Thesis Artifacts.” [Online]. Available: https://doi.org/10.5281/zenodo.17116089

[35] “GCC git repository - Initial Commit.” [Online]. Available: https://gcc.gnu.org/git/?p=gcc.git;a=tree;h=f2fcf612cfc29038ed6a7196ce31eb05c80ab094;hb=0963cb5fde158cce986523a90fa9edc51c881f31

[36] FOSDEM Organizers, “FOSDEM 2024 Website.” [Online]. Available: https://archive.fosdem.org/2024/

[37] Pierrick Philippe, “Unlocking Secret Analysis in GCC Static Analyzer.” [Online]. Available: https://archive.fosdem.org/2024/schedule/event/fosdem-2024-2635-unlocking-secret-analysis-in-gcc-static-analyzer/

[38] “GCC Static Analyzer Evaluation Artifacts.” [Online]. Available: https://doi.org/10.5281/zenodo.17116293

[39] “GnuZero Artifacts.” [Online]. Available: https://doi.org/10.5281/zenodo.14277842

[40] Anonym, “GnuSecret artifacts.” [Online]. Available: https://doi.org/10.5281/zenodo.16106526

[41] Richard M. Stallman and the GCC Developer Community, “GCC Internal Documentation.” [Online]. Available: https://gcc.gnu.org/onlinedocs/gccint.pdf

[42] J. Merrill, “GENERIC and GIMPLE: A New Tree Representation for Entire Functions,” in *GCC Summit*, 2003.

[43] D. Novillo, “Design and Implementation of Tree SSA,” in *GCC Summit*, 2004.

[44] D. Novillo, “Tree SSA: A New Optimization Infrastructure for GCC,” in *GCC Summit*, 2003.

[45] J. Hubička, “Interprocedural optimization on function local SSA form in GCC,” in *GCC Summit*, 2006.

[46] L. J. Hendren, C. Donawa, M. Emami, G. R. Gao, Justiani, and B. Sridharan, "Designing the McCAT Compiler Based on a Family of Structured Intermediate Representations," in *LCPC*, in Lecture Notes in Computer Science, vol. 757. Springer, 1992, pp. 406–420.

[47] B. Alpern, M. N. Wegman, and F. K. Zadeck, "Detecting Equality of Variables in Programs," in *POPL*, ACM Press, 1988, pp. 1–11.

[48] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global Value Numbers and Redundant Computations," in *POPL*, ACM Press, 1988, pp. 12–27.

[49] C. Lattner and V. S. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *CGO*, IEEE Computer Society, 2004, pp. 75–88.

[50] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph," *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 4, pp. 451–490, 1991.

[51] "SSA - Emails exchanged on GCC's mailing list." [Online]. Available: https://gcc.gnu.org/pipermail/gcc/2023-March/240894.html

[52] F. Wiedijk, "C23 standard - N3220 working draft." [Online]. Available: https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3220.pdf

[53] LLVM development community, "LLVM Language Reference manual." [Online]. Available: https://llvm.org/docs/LangRef.html

[54] R. Morgan, *Building an Optimizing Compiler*. Butterworth, 1998.

[55] J. W. Davidson and C. W. Fraser, "The Design and Application of a Retargetable Peephole Optimizer," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 2, pp. 191–202, 1980.

[56] R. E. Johnson, C. McConnell, and J. M. Lake, "The RTL System: A Framework for Code Optimization," in *Code Generation*, in Workshops in Computing. Springer, 1991, pp. 255–274.

[57] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand, "CompCert - A Formally Verified Optimizing Compiler," in *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France, Jan. 2016. [Online]. Available: https://inria.hal.science/hal-01238879

[58] Richard M. Stallman and the GCC Developer Community, "GCC User Documentation." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc.pdf

[59] D. Svoboda, "Towards support for attributes in C." [Online]. Available: https://www.open-std.org/JTC1/SC22/WG14/www/docs/n1403.pdf

[60] The kernel development community, "Programming Language." [Online]. Available: https://docs.kernel.org/5.10/process/programming-language.html

[61] A. Ballman, "Aaron Ballman Website." [Online]. Available: https://aaronballman.com/

[62] The kernel development community, "Linux Build System Documentation." [Online]. Available: https://www.kernel.org/doc/html/latest/kbuild/

[63] GRSecurity, "GRSecurity Website." [Online]. Available: https://grsecurity.net/

[64] PaX, "PaX Team Website." [Online]. Available: https://pax.grsecurity.net/

[65] The Linux Development Community, "Linux - GitHub Repository." [Online]. Available: https://github.com/torvalds/linux

[66] M. Sebor, "Understanding GCC warnings, Part 2." [Online]. Available: https://developers.redhat.com/blog/2019/03/13/understanding-gcc-warnings-part-2

[67] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 1953, doi: 10.1090/s0002-9947-1953-0053041-6.

[68] B. Livshits *et al.*, "In defense of soundiness: a manifesto," *Commun. ACM*, vol. 58, no. 2, pp. 44–46, 2015.

[69] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-C: A software analysis perspective," *Formal Aspects Comput.*, vol. 27, no. 3, pp. 573–609, 2015.

[70] Facebook, “Infer Static Analyzer.” [Online]. Available: https://fbinfer.com/

[71] “Coverity Scan (Synopsys) - Static Analysis.” [Online]. Available: https://scan.coverity.com/

[72] P. Cousot *et al.*, “The ASTREÉ Analyzer,” in *ESOP*, in Lecture Notes in Computer Science, vol. 3444. Springer, 2005, pp. 21–30.

[73] A. Djoudi and S. Bardin, “BINSEC: Binary Code Analysis with Low-Level Regions,” in *TACAS*, in Lecture Notes in Computer Science, vol. 9035. Springer, 2015, pp. 212–217.

[74] R. S. Boyer, B. Elspas, and K. N. Levitt, “SELECT - a formal system for testing and debugging programs by symbolic execution,” in *Reliable Software*, ACM, 1975, pp. 234–245.

[75] J. C. King, “Symbolic Execution and Program Testing,” *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[76] W. E. Howden, “Experiments with a symbolic evaluation system,” in *AFIPS National Computer Conference*, in AFIPS Conference Proceedings, vol. 45. AFIPS Press, 1976, pp. 899–908.

[77] L. A. Clarke, “A program testing system,” in *ACM Annual Conference*, ACM, 1976, pp. 488–491.

[78] J. C. King, “A New Approach to Program Testing,” in *Programming Methodology*, in Lecture Notes in Computer Science, vol. 23. Springer, 1974, pp. 278–290.

[79] “Z3Prover.” [Online]. Available: https://github.com/Z3Prover/z3

[80] “Boolector Website.” [Online]. Available: https://boolector.github.io/

[81] C. Cadar and K. Sen, “Symbolic execution for software testing: three decades later,” *Commun. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[82] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, “A Survey of Symbolic Execution Techniques,” *ACM Comput. Surv.*, vol. 51, no. 3, pp. 1–39, 2018.

[83] Z. Xu, T. Kremenek, and J. Zhang, “A Memory Model for Static Analysis of C Programs,” in *ISoLA (1)*, in Lecture Notes in Computer Science, vol. 6415. Springer, 2010, pp. 535–548.

[84] C. Cadar, D. Dunbar, and D. R. Engler, “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs,” in *OSDI*, USENIX Association, 2008, pp. 209–224.

[85] Y. Yuan, Z. Zhou, J. Belyakova, and S. Jagannathan, “Derivative-Guided Symbolic Execution,” *Proc. ACM Program. Lang.*, vol. 9, no. POPL, pp. 1475–1505, 2025.

[86] P. Cousot and R. Cousot, “Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints,” in *POPL*, ACM, 1977, pp. 238–252.

[87] B. Blanchet *et al.*, “A static analyzer for large safety-critical software,” in *PLDI*, ACM, 2003, pp. 196–207.

[88] D. Bühler, “Structuring an Abstract Interpreter through Value and State Abstractions:EVA, an Evolved Value Analysis for Frama-C. (Structurer un interpréteur abstrait au moyen d'abstractions de valeurs et d'états :Eva, une analyse de valeur évoluée pour Frama-C),” 2017.

[89] G. A. Kildall, “A Unified Approach to Global Program Optimization,” in *POPL*, ACM Press, 1973, pp. 194–206.

[90] E. J. Schwartz, T. Avgerinos, and D. Brumley, “All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask),” in *IEEE Symposium on Security and Privacy*, IEEE Computer Society, 2010, pp. 317–331.

[91] N. Jovanovic, C. Krügel, and E. Kirda, “Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper),” in *S&P*, IEEE Computer Society, 2006, pp. 258–263.

[92] A. Avancini and M. Ceccato, “Towards security testing with taint analysis and genetic algorithms,” in *SESS@ICSE*, ACM, 2010, pp. 65–71.

[93] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, “DR. CHECKER: A Soundy Analysis for Linux Kernel Drivers,” in *USENIX Security Symposium*, USENIX Association, 2017, pp. 1007–1024.

[94] A. W. Marashdih, Z. F. Zaaba, and K. Suwais, “An Enhanced Static Taint Analysis Approach to Detect Input Validation Vulnerability,” *J. King Saud Univ. Comput. Inf. Sci.*, vol. 35, no. 2, pp. 682–701, 2023.

[95] M. M. Hossain, F. Farahmandi, M. Tehranipoor, and F. Rahman, "BOFT: Exploitable Buffer Overflow Detection by Information Flow Tracking," in *DATE*, IEEE, 2021, pp. 1126–1129.

[96] Multiple Contributors, "SEI CERT C Coding Standard - Taint Analysis." [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/c/Taint+Analysis

[97] L. P. Cox *et al.*, "SpanDex: Secure Password Tracking for Android," in *USENIX Security Symposium*, USENIX Association, 2014, pp. 481–494.

[98] Lars Ole Andersen, "Program Analysis and Specialization for the C Programming Language," 1994. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=b7efe971a34a0f2482e0b2520ffb31062dcdde62

[99] B. Steensgaard, "Points-to Analysis in Almost Linear Time," in *POPL*, ACM Press, 1996, pp. 32–41.

[100] C. Lattner, A. Lenharth, and V. S. Adve, "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *PLDI*, ACM, 2007, pp. 278–289.

[101] B. Hardekopf and C. Lin, "Exploiting Pointer and Location Equivalence to Optimize Pointer Analysis," in *SAS*, in Lecture Notes in Computer Science, vol. 4634. Springer, 2007, pp. 265–280.

[102] E. M. Clarke, O. Grumberg, and D. E. Long, "Model Checking and Abstraction," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 5, pp. 1512–1542, 1994.

[103] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, 1986.

[104] E. M. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in *TACAS*, in Lecture Notes in Computer Science, vol. 2988. Springer, 2004, pp. 168–176.

[105] E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," *Formal Methods Syst. Des.*, vol. 19, no. 1, pp. 7–34, 2001.

[106] T. Coquand and G. P. Huet, "The Calculus of Constructions," *Inf. Comput.*, vol. 76, no. 2/3, pp. 95–120, 1988.

[107] T. Ball, "The Concept of Dynamic Analysis," in *ESEC / SIGSOFT FSE*, in Lecture Notes in Computer Science, vol. 1687. Springer, 1999, pp. 216–234.

[108] J. R. Larus and T. Ball, "Rewriting Executable Files to Measure Program Behavior," *Softw. Pract. Exp.*, vol. 24, no. 2, pp. 197–218, 1994.

[109] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *USENIX ATC*, USENIX Association, 2012, pp. 309–318.

[110] E. Stepanov and K. Serebryany, "MemorySanitizer: fast detector of uninitialized memory use in C++," in *CGO*, IEEE Computer Society, 2015, pp. 46–55.

[111] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, ACM, 2007, pp. 89–100.

[112] B. P. Miller, L. Fredriksen, and B. So, "An Empirical Study of the Reliability of UNIX Utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[113] Zalewski Michal (lcamtuf), "American Fuzzy Lop (AFL)." [Online]. Available: https://github.com/google/AFL

[114] A. Fioraldi, A. Mantovani, D. C. Maier, and D. Balzarotti, "Dissecting American Fuzzy Lop: A FuzzBench Evaluation," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, pp. 1–26, 2023.

[115] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based Greybox Fuzzing as Markov Chain," in *CCS*, ACM, 2016, pp. 1032–1043.

[116] A. Hazimeh, A. Herrera, and M. Payer, "Magma: A Ground-Truth Fuzzing Benchmark," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 4, no. 3, pp. 1–29, 2020.

[117] A. Fioraldi, D. C. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining Incremental Steps of Fuzzing Research," in *WOOT @ USENIX Security Symposium*, USENIX Association, 2020.

[118] "Secretgrind: a Valgrind Analysis Tool to Detect Secrets in Memory." [Online]. Available: https://github.com/lmrs2/secretgrind

[119] K. Gudka *et al.*, "Clean Application Compartmentalization with SOAAP," in *CCS*, ACM, 2015, pp. 1016–1031.

[120] Wei Ming Khoo, "A Taint-Tracking Plugin for the Valgrind Memory Checking Tool." [Online]. Available: https://github.com/wmkhoo/taintgrind

[121] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *PLDI*, ACM, 2005, pp. 213–223.

[122] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *ESEC/SIGSOFT FSE*, ACM, 2005, pp. 263–272.

[123] S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!," in *USENIX Security Symposium*, USENIX Association, 2020, pp. 181–198.

[124] The angr Development Community, "angr." [Online]. Available: https://angr.io/

[125] N. Stephens *et al.*, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution," in *NDSS*, The Internet Society, 2016.

[126] A. Mantovani, A. Fioraldi, and D. Balzarotti, "Fuzzing with Data Dependency Information," in *EuroS&P*, IEEE, 2022, pp. 286–302.

[127] Github - LLVM Project, "Clang's static analyzer should be able to warn when memset()/memmove()/memcpy() are optimized away by dead store elimination." [Online]. Available: https://github.com/llvm/llvm-project/issues/61098

[128] E. B. Sørensen, E. K. Karlsen, and J. Li, "What Norwegian Developers Want and Need From Security-Directed Program Analysis Tools: A Survey," in *EASE*, ACM, 2020, pp. 505–511.

[129] A. Bessey *et al.*, "A few billion lines of code later: using static analysis to find bugs in the real world," *Commun. ACM*, vol. 53, no. 2, pp. 66–75, 2010.

[130] T. W. Reps, S. Horwitz, and S. Sagiv, "Precise Interprocedural Dataflow Analysis via Graph Reachability," in *POPL*, ACM Press, 1995, pp. 49–61.

[131] C. S. Strachey, "Fundamental Concepts in Programming Languages," *High. Order Symb. Comput.*, vol. 13, no. 1/2, pp. 11–49, 2000.

[132] Free Software Foundation, "GCC Static Analyzer Documentation." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Static-Analyzer-Options.html

[133] D. Malcolm, "The Malloc State Machine." [Online]. Available: https://dmalcolm.fedorapeople.org/gcc/2019-11-22/sm-malloc.png

[134] Multiple Contributors, "SEI CERT C Coding Standard - POSIX recommendation." [Online]. Available: https://wiki.sei.cmu.edu/confluence/display/c/POS34-C.+Do+not+call+putenv%28%29+with+a+pointer+to+an+automatic+variable+as+the+argument

[135] SAMATE project - Software Quality Group, "NIST Software Assurance Reference Dataset." [Online]. Available: https://samate.nist.gov/SARD/

[136] OpenSSL development community, "OpenSSL's security advisory for CVE-2020-1967." [Online]. Available: https://openssl-library.org/news/secadv/20200421.txt

[137] irsl (https://github.com/irsl), "CVE-2020-1967 Exploit." [Online]. Available: https://github.com/irsl/CVE-2020-1967

[138] D. Peter, "hyperfine." [Online]. Available: https://github.com/sharkdp/hyperfine

[139] S. A. Olmos *et al.*, "High-assurance zeroization," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, vol. 2024, no. 1, pp. 375–397, 2024.

[140] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding Data Lifetime via Whole System Simulation (Awarded Best Paper!)," in *USENIX Security Symposium*, USENIX, 2004, pp. 321–336.

[141] R. Chapman, "Sanitizing Sensitive Data: How to Get It Right (or at Least Less Wrong...)," in *Ada-Europe*, in Lecture Notes in Computer Science, vol. 10300. Springer, 2017, pp. 37–52.

[142] The MITRE corporation, "CWE-14: Compiler Removal of Code to Clear Buffers." [Online]. Available: https://cwe.mitre.org/data/definitions/14.html

[143] Z. Yang, B. Johannesmeyer, A. T. Olesen, S. Lerner, and K. Levchenko, "Dead Store Elimination (Still) Considered Harmful," in *USENIX Security Symposium*, USENIX Association, 2017, pp. 1025–1040.

[144] C. Percival, "Erratum." [Online]. Available: https://www.daemonology.net/blog/2014-09-05-erratum.html

[145] M. Howard, "Some Bad News and Some Good News." [Online]. Available: https://learn.microsoft.com/en-us/previous-versions/ms972826(v=msdn.10)

[146] E. Eide and J. Regehr, "Volatiles are miscompiled, and what to do about it," in *EMSOFT*, ACM, 2008, pp. 255–264.

[147] Github - Bitcoin Core, "Memory zeroization improvements." [Online]. Available: https://github.com/bitcoin-core/secp256k1/issues/185

[148] Github - Linux PAM, "Memory erasure followups." [Online]. Available: https://github.com/linux-pam/linux-pam/pull/599

[149] National Vulnerability Database, "CVE-2023-32784 Detail." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-32784

[150] MITRE, [Online]. Available: https://www.mitre.org/

[151] Free Software Foundation, "GCC Static Analyzer Documentation." [Online]. Available: https://sourceware.org/glibc/wiki/MallocInternals#Realloc_Algorithm

[152] Multiple Contributors, "SEI CERT C Coding Standard - MEM03-C. Clear sensitive information stored in reusable resources." [Online]. Available: https://wiki.sei.cmu.edu/confluence/x/VNcxBQ

[153] ANSSI, "Rules for Secure C Language Software Development." [Online]. Available: https://cyber.gouv.fr/sites/default/files/2022/04/anssi-guide-rules_for_secure_c_language_software_development-v1.4.pdf#section*.162

[154] GCC Online Documentation, "Common Function Attributes." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Common-Function-Attributes.html#index-tainted_005fargs-function-attribute

[155] R. David *et al.*, "BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis," in *SANER*, IEEE Computer Society, 2016, pp. 653–656.

[156] D. J. Wheeler and R. M. Needham, "TEA, a Tiny Encryption Algorithm," in *FSE*, in Lecture Notes in Computer Science, vol. 1008. Springer, 1994, pp. 363–366.

[157] D. J. Wheeler and R. M. Needham, "Tea extensions." [Online]. Available: https://www.cix.co.uk/~klockstone/xtea.pdf

[158] WPA Supplicant Development Community, "wpa_supplicant_set_wpa_none_key - GitHub Repository." [Online]. Available: https://github.com/michael-dev/hostapd/blob/f91680c15f80f0b617a0d2c369c8c1bb3dcf078b/wpa_supplicant/wpa_supplicant.c#L158

[159] J. Chow, B. Pfaff, T. Garfinkel, and M. Rosenblum, "Shredding Your Garbage: Reducing Data Lifetime Through Secure Deallocation," in *USENIX Security Symposium*, USENIX Association, 2005.

[160] R. R. Hansen and C. W. Probst, "Non-Interference and Erasure Policies for Java Card Bytecode," in *6th International Workshop on Issues in the Theory of Security (WITS '06)*, USENIX Association, 2006.

[161] A. Pridgen, S. L. Garfinkel, and D. S. Wallach, "Present but Unreachable: Reducing Persistentlatent Secrets in HotSpot JVM," in *HICSS*, ScholarSpace / AIS Electronic Library (AISeL), 2017, pp. 1–10.

[162] J. Lee and D. S. Wallach, "Removing Secrets from Android's TLS," in *NDSS*, The Internet Society, 2018.

[163] J. Lee, A. Chen, and D. S. Wallach, "Total Recall: Persistence of Passwords in Android," in *NDSS*, The Internet Society, 2019.

[164] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[165] L. Simon, D. Chisnall, and R. J. Anderson, "What You Get is What You C: Controlling Side Effects in Mainstream C Compilers," in *EuroS&P*, IEEE, 2018, pp. 1–15.

[166] S. T. Vu, A. Cohen, A. de Grandmaison, C. Guillon, and K. Heydemann, "Reconciling optimization with secure compilation," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1–30, 2021.

[167] GCC Online Documentation, "Common Type Attributes." [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Common-Type-Attributes.html#index-strub-type-attribute

[168] Y. Tang, P. Ames, S. Bhamidipati, A. Bijlani, R. Geambasu, and N. Sarda, "CleanOS: Limiting Mobile Data Exposure with Idle Eviction," in *OSDI*, USENIX Association, 2012, pp. 77–91.

[169] T. Müller, F. C. Freiling, and A. Dewald, "TRESOR Runs Encryption Securely Outside RAM," in *USENIX Security Symposium*, USENIX Association, 2011.

[170] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "CaSE: Cache-Assisted Secure Execution on ARM Processors," in *IEEE Symposium on Security and Privacy*, IEEE Computer Society, 2016, pp. 72–90.

[171] M. Kolosick *et al.*, "Robust Constant-Time Cryptography," *CoRR*, 2023.

[172] Y. Du, O. Alrawi, K. Z. Snow, M. Antonakakis, and F. Monrose, "Improving Security Tasks Using Compiler Provenance Information Recovered At the Binary-Level," in *CCS*, ACM, 2023, pp. 2695–2709.

[173] W. Enck *et al.*, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," in *OSDI*, USENIX Association, 2010, pp. 393–407.

[174] Jeremy Rifkin, "Wyrm: a GIMPLE to LLVM IR transpiler." [Online]. Available: https://github.com/jeremy-rifkin/wyrm

[175] S. Cauligi *et al.*, "Constant-time foundations for the new spectre era," in *PLDI*, ACM, 2020, pp. 913–926.

[176] PLSysSec, "Github - Haybale." [Online]. Available: https://github.com/PLSysSec/haybale

[177] D. E. Denning and P. J. Denning, "Certification of Programs for Secure Information Flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[178] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: effective taint analysis of web applications," in *PLDI*, ACM, 2009, pp. 87–97.

[179] K. Ashcraft and D. R. Engler, "Using Programmer-Written Compiler Extensions to Catch Security Holes," in *S&P*, IEEE Computer Society, 2002, pp. 143–159.

[180] L. Cai, F. Song, and T. Chen, "Towards Efficient Verification of Constant-Time Cryptographic Implementations," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1019–1042, 2024.

[181] M. Häuser and K. Schneider, "Secret Types Require OS-Backed Secrecy Code Sections," in *MoVe4SPS*, 2025.

[182] A. Johansson, C. Holmberg, F. G. de Oliveira Neto, and P. Leitner, "The Impact of Compiler Warnings on Code Quality in C++ Projects," in *ICPC*, ACM, 2024, pp. 270–279.

[183] V. Yodaiken, "How ISO C became unusable for operating systems development," *CoRR*, 2022.

[184] V. D'Silva, M. Payer, and D. X. Song, "The Correctness-Security Gap in Compiler Optimization," in *IEEE Symposium on Security and Privacy Workshops*, IEEE Computer Society, 2015, pp. 73–87.

[185] U. Shankar, K. Talwar, J. S. Foster, and D. A. Wagner, "Detecting Format String Vulnerabilities with Type Qualifiers," in *USENIX Security Symposium*, USENIX, 2001.

[186] N. Broberg, B. van Delft, and D. Sands, "The Anatomy and Facets of Dynamic Policies," in *CSF*, IEEE Computer Society, 2015, pp. 122–136.

[187] D. Book, "Security Mechanisms and Concerns." [Online]. Available: https://perldoc.perl.org/perlsec

[188] Z. community, "Zig GitHub - Issue on Constant Time." [Online]. Available: https://github.com/ziglang/zig/issues/1776

[189] GCC, "GCC 14 Release Series – Changes, New Features, and Fixes." [Online]. Available: https://gcc.gnu.org/gcc-14/changes.html

[190] M. Abadi, "Secrecy by Typing inSecurity Protocols," in *TACS*, in Lecture Notes in Computer Science, vol. 1281. Springer, 1997, pp. 611–638.

[191] National Vulnerability Database, "CVE-2023-5981 Detail." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-5981

[192] National Vulnerability Database, "CVE-2023-25189 Detail." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2024-25189

[193] National Vulnerability Database, "CVE-2023-5992 Detail." [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-5992

[194] IETF, "JWS RFC." [Online]. Available: https://datatracker.ietf.org/doc/html/rfc7515

[195] D. Bleichenbacher, "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1," in *CRYPTO*, in Lecture Notes in Computer Science, vol. 1462. Springer, 1998, pp. 1–12.

[196] GCC, "GCC 13 Release Series – Changes, New Features, and Fixes." [Online]. Available: https://gcc.gnu.org/gcc-13/changes.html

[197] S. Chen, Z. Lin, and Y. Zhang, "SelectiveTaint: Efficient Data Flow Tracking With Static Binary Rewriting," in *USENIX Security Symposium*, USENIX Association, 2021, pp. 1665–1682.

[198] C. Wang, R. Ko, Y. Zhang, Y. Yang, and Z. Lin, "Taintmini: Detecting Flow of Sensitive Data in Mini-Programs with Static Taint Analysis," in *ICSE*, IEEE, 2023, pp. 932–944.

[199] L. Zhang, J. Chen, W. Diao, S. Guo, J. Weng, and K. Zhang, "CryptoREX: Large-scale Analysis of Cryptographic Misuse in IoT Devices," in *RAID*, USENIX Association, 2019, pp. 151–164.

[200] D. A. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," in *NDSS*, The Internet Society, 2000.

[201] K. Cheng *et al.*, "DTaint: Detecting the Taint-Style Vulnerability in Embedded Device Firmware," in *DSN*, IEEE Computer Society, 2018, pp. 430–441.

[202] J. Zhao *et al.*, "Leveraging Semantic Relations in Code and Data to Enhance Taint Analysis of Embedded Systems," in *USENIX Security Symposium*, USENIX Association, 2024.

[203] S. A. Olmos *et al.*, "Let's DOIT: Using Intel's Extended HW/SW Contract for Secure Compilation of Crypto Code," *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025.

[204] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying Constant-Time Implementations," in *USENIX Security Symposium*, USENIX Association, 2016, pp. 53–70.

[205] C. Disselkoen, "pitchfork: Verifying constant-time code with symbolic execution." [Online]. Available: https://github.com/PLSysSec/haybale-pitchfork

[206] M. Neikes, "Automated dynamic analysis for timing side-channels." [Online]. Available: https://www.post-apocalyptic-crypto.org/timecop/

[207] J. B. Almeida *et al.*, "Jasmin: High-Assurance and High-Speed Cryptography," in *CCS*, ACM, 2017, pp. 1807–1823.

[208] A. Geimer, M. Vergnolle, F. Recoules, L.-A. Daniel, S. Bardin, and C. Maurice, "A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries," in *CCS*, ACM, 2023, pp. 1690–1704.

[209] M. Fourné *et al.*, ""These results must be false": A usability evaluation of constant-time analysis tools," *IACR Cryptol. ePrint Arch.*, p. 2060, 2024.

[210] M. Schneider, D. Lain, I. Puddu, N. Dutly, and S. Capkun, "Breaking Bad: How Compilers Break Constant-Time~Implementations," *CoRR*, 2024.

[211] L. Gerlach, R. Pietsch, and M. Schwarz, "Do Compilers Break Constant-time Guarantees?," in *Financial Cryptography (1)*, in Lecture Notes in Computer Science. Springer, 2025.

[212] T. Faingnaert, W. V. Iseghem, and B. D. Sutter, "K-Hunt++: Improved Dynamic Cryptographic Key Extraction," in *Checkmate@CCS*, ACM, 2024, pp. 22–29.

[213] C. Meijer, V. Moonsamy, and J. Wetzels, "Where's Crypto?: Automated Identification and Classification of Proprietary Cryptographic Primitives in Binary Code," in *USENIX Security Symposium*, USENIX Association, 2021, pp. 555–572.

COLLEGE MATHS, TELECOMS
DOCTORAL INFORMATIQUE, SIGNAL
BRETAGNE SYSTEMES, ELECTRONIQUE

Université de Rennes

Titre : Secrets dans le Compilateur: Détection de Faiblesses Liées au Secret dans l'Analyseur Statique de GCC

Mots clés : Analyse Statique, GCC, Analyseur Statique de GCC, Teinte, Mise à Zéro, GnuZero, GnuSecret

Résumé : Les systèmes logiciels sont omniprésents dans les sociétés modernes. Qu'il s'agisse de smartphones ou de services publics essentiels, nombre d'entre eux traitent des données sensibles, rendant critique leur protection contre les fuites et les erreurs de gestion de ces données. Parmi les faiblesses récurrentes, comme l'illustrent la CWE-200 et ses descendants, la divulgation involontaire d'information est une menace persistante. Cette thèse explore comment l'analyse statique intégrée au sein d'un compilateur peut détecter automatiquement de telles faiblesses. Nous commençons par disséquer le GCC Static Analyzer (GSA), en documentant son moteur d'exécution symbolique, son modèle mémoire, et son système de diagnostique, le mettant ainsi en évidence comme un bon candidat pour des analyses de sécurité. En nous appuyant sur cette base, nous proposons GnuZero, un outil basé sur GCC pour détecter des mises à zéro manquantes de données sensibles, identifiées par les CWE-226 et CWE-244, grâce à un ensemble d'attributs dédiés et d'une analyse de propagation de teinte basée sur la durée de vie des variables, validée sur des bancs d'essai et des logiciels du monde réel. Nous généralisons ensuite cette approche avec GnuSecret, un cadriciel unifiant le suivi de secrets pour capturer des faiblesses plus larges de divulgation d'informations, détectant avec succès des vulnérabilités connues et nouvelles. Ensemble, ces contributions démontrent la faisabilité et l'efficacité des analyses mises en œuvre dans un compilateur largement utilisé.

Title: Secrets in Compiler: Detection of Secret-related Weaknesses in GCC Static Analyzer

Keywords: Static Analysis, GCC, GCC Static Analyzer, Tainting, Zeroization, GnuZero, GnuSecret

Abstract: Software systems are ubiquitous in modern societies. From smartphones to critical state services, many handle sensitive data, making their protection against leaks and improper management critical. Among recurring weaknesses, as illustrated by CWE-200 and its descendants, unintended information disclosure remains a persistent threat. This thesis explores how compiler-integrated static analysis can automatically detect such vulnerabilities. We first dissect the GCC Static Analyzer (GSA), documenting its symbolic execution engine, memory model, and reporting system, thereby highlighting it as a good fit for security-oriented analyses. Building on this groundwork, we propose GnuZero, a GCC-based tool to detect missing zeroization of sensitive data, identified by CWE-226 and CWE-244, through dedicated attributes and taint-propagation analysis based on variables' lifetime, validated on benchmarks and real-world software. We then generalize this approach with GnuSecret, a framework unifying secrecy tracking to capture broader information disclosure weaknesses, successfully detecting both known and new vulnerabilities. Together, these contributions demonstrate the feasibility and effectiveness of analyses implemented within a broadly used compiler.