Securing system code with minimal developer efforts: from preventing type confusion by design to automated library fuzzing

THIS IS A TEMPORARY TITLE PAGE
It will be replaced for the final print by a version
provided by the registrar's office.

Doctoral Thesis
by Nicolas Badoux

presented on 7th April, 2024
at EPFL, Lausanne

Pending approbation by the examining committee:

Prof. Anne-Marie Kermarrec, *Jury President*
Prof. Mathias Payer, *Thesis Director*
Prof. Thomas Bourgeat, *Internal Examiner*
Prof. Alessandra Gorla, *External Examiner*
Dr. Nathan Burow, *External Examiner*

If I have the gift of prophecy and can fathom all mysteries and all knowledge, and if I have a faith that can move mountains, but do not have love, I am nothing.

First Corinthians 13:2—The Bible

To my daughter, Coline, may your life be filled with love, joy, and the gift of faith.
To my wife, Marie, for her unwavering support and communicative joy.

Acknowledgments

A PhD journey is filled with unexpected turns and challenging dead-ends. As I approach its conclusion, I am deeply grateful to all those who supported me throughout these years.

First and foremost, I want to acknowledge my faith in Jesus Christ that has been my foundation throughout this journey. His unconditional love and grace have given me the confidence and joy to pursue this PhD to the end. I have strived to reflect those gifts around me, and I pray to continue doing so beyond these academic years.

I am especially grateful to Prof. Mathias Payer for offering me this PhD opportunity. Your guidance, constant encouragement, and support—both academic and extra-curricular—have been invaluable. Your down-to-earth attitude, openness, and humor made this journey not just enriching but enjoyable. I particularly admire your dedication to the wellbeing of HexHive members and your family. I also extend my sincere thanks to my thesis committee members—Prof. Anne-Marie Kermarrec, Prof. Thomas Bourgeat, Prof. Alessandra Gorla, and Dr. Nathan Burow—for their time, insights, and constructive feedback that helped shape this thesis.

Throughout my PhD, I witnessed many arrivals and departures in HexHive. When asked about my work, I always emphasized our group's diversity, talent, and vibrant atmosphere that made coming to the office a joy. A special thanks goes to then Dr. and now Prof. Flavio Toffalini. You joined HexHive when type++ was at its lowest point, and through your support, constant positivity, and humor, we elevated it to a distinguished paper! While I couldn't turn you into an active skier, you will always be welcome for a fondue. Long live PizzaLab!

After COVID-19, I had the pleasure of sharing an office with Dr. Ahmad Hazimeh. Our enduring legacy to BC154 will be the fan we *borrowed* from the archive room. Our discussions about Lebanon, your delicious cannelés, and your daily bike repair updates made each day special. After Ahmad departure, there was continuity as Dr. Qiang Liu showed up with a new bike every two months. Your insights about China and your thrilling apartment saga kept me on edge daily.

HexHive outings gained a special flavor thanks to Dr. Atri Bhattacharyya. Despite the Zug immigration office's skepticism, your passion for the Swiss Alps and dedication to exploring every via ferrata and hiking trail make you the swisss-est Indian I know—looking forward to your naturalization party!

It has been my great pleasure to work with colleagues from all over the world. Understanding their diverse cultures and worldviews became a driving force in my daily quest for motivation to show up at the office. I particularly enjoyed our Serbian crew—Jelena Janković, Uroš Tešić (and his beloved owl!), and Nevenka Savić. Through our visiting students, I discovered China's rich culture, starting with Prof. Zhiyuan Jiang and continuing with Jianhao Xu, Tao Lyu, Zezhong Ren, Zheyu Ma, Duo Xu, and Yishun Zeng. I'm grateful to the HexHive members who regularly organized Chinese hotpot gatherings, and especially to Zhiyao Feng, whose weekly food orders helped me master the chopsticks. Chibin Zhang and Qinying Wang's thoughtful lunch-time knocks were always welcome breaks. Beyond the culinary adventures, Han Zheng's technical support—whether

rebooting servers outside office hours or sharing fuzzing expertise—proved invaluable. I look forward to experiencing China firsthand and putting my WeChat account to good use!

While I was impressed by our Chinese visitors' commitment to learn skiing, Marcel Busch topped them all with his rapid progress on the slopes and enthusiasm for full-day skiing sessions. Thank you also for your feedback and support throughout your post-doc. Each morning, passing by Luca Di Bartolomeo and Florian Hofhammer's office was an adventure in spotting what was new: another sticker, a fresh crate of Maté, or a new plush toy. Florian's unwavering commitment to a clean working space likely saved us from numerous pandemics. Your helpfulness and good spirits made life in HexHive a joy! While guessing if Luca was in the office remained a daily challenge, I cherish our meaningful life discussions! I would also like to acknowledge the newest additions to HexHive: Solène Husseini, Zurab Tsinadze, Dariia Kharytonova, Rokhaya Fall, and Rafaila Galanopoulou. I wish you the best of luck in your PhD journeys and hope you will enjoy your time at HexHive as much as I did (and not only because I am vacating an office). To finish with the European crew, while I started my PhD as the token Swiss, I am happy that Philipp Mao and Mathias will be able to push for more fondue and raclette at the different HexHive events!

Beyond the current HexHive members, my journey was enriched by many who came before. I am grateful to Andrès Sanchez, whose humor brightened our days; Lucio Romerio for his kindness and hospitality; Gwangmu Lee for his genuine curiosity about Swiss culture; Derrick McKee for bringing laughter to our virtual meetings; Antony Vennard for sharing his C++ expertise and love for Switzerland; Hui Peng for his invaluable guidance during my Master's thesis; and Jean-Michel Crepel for adding that quintessential French flair to our office.

I would also like to express my heartfelt gratitude to Natacha Fontana, our dedicated administrative assistant. Your steadfast support throughout my PhD journey, your insightful perspectives on both our school and research group, and above all, your genuine kindness and thoughtfulness toward me and my family have been truly outstanding.

Outside of HexHive, my time at EPFL has been enriched by meaningful friendships, particularly with Julien Lamour (and by extension Marta) and Nicolas Roussel. While the Lamborghini remains a dream, the hold-up strategy seems to have worked once again. Thank you for your presence throughout these years and for all the cherished memories we've created. Nicolas, I will miss our lunches, but who knows where life will take us—perhaps to that hut in the Pyrénées. Additionally, special thanks to Marc Illunga for all these years of friendship!

As my colleagues can testify, my office was empty on weekends, thanks to the many friend circles I am blessed with. I'm grateful to the Interjeunes community for welcoming me long past my youth—the diversity of professional paths, ages, and friendships continues to enrich my life. Though less present at GBEU during my PhD than during my Bachelor's, the friendships formed there and our deep discussions have shaped my worldview and faith. May God continue to reign on our campus and in our hearts! The Sport & Aventure children's camps were yearly highlights and defining moments for my couple. My childhood friends from la Vallée de Joux remain dear to me (special mention to Blaise Robert, who always foudn time to say hi when back from Boston and my goddaughter, Svenja Rochat, who grew so much during these years). The lifelong friendships

with Rémy Wagner (extended to Noémie) and Luc-Antoine Badoux have been foundational—may the quilles and fondue continue to bring us joy! Finally, I want to thank Marc Zürcher, whose friendship and confidence proved invaluable during the inevitable low points 🦄.

Throughout these five years, my roommates—Elisa Meylan, Céline Boutay, Sylvain Kaufmann, Estelle Guex, Anne-Sophie Zuber, and Jérémie Bidaux—endured my late working hours and the paper rejections. Your friendships, kindness, and warmth helped me navigate this journey! A special thanks also to the summer vacation crew for the memorable breaks from research.

Lastly, I would like to thank my family. Starting with the extended one who offered me advice and graciously accommodated my work during our beloved vacations. Second, my in-laws who are a source of joy and support. Third, my grandparents: Antoinette Gilliand for her kindness, joy, and affection; Daniel for his courage and entrepreneurial spirit; Eliane Badoux for her hospitality and compassion; and Jean-Claude for his work ethic and academic pursuits. Most importantly, they all showed me how faith in Christ can shape a person and its environment for the better. I am deeply thankful to my siblings and their spouses—Guillaume and Rahel, Joanne and Ivanoé, and Maëlle–for their support, prayers, and encouragement. Finally, to my parents, Luc and Yolande, you invested in me long before I have any memories. The love you showed me throughout my life allowed me to grow confident in my abilities, curious about the world, and steadfast in my faith in Jesus Christ. Thank you for your patience with my shortcomings and for enabling me to pursue this PhD in the best possible conditions. While I can never repay you, I hope to be able to offer the same love to your grandchildren.

Coline, you are the greatest gift this PhD journey has given me. I could spend hours watching your cheerful smile and curious eyes explore the world. I look forward to seeing you grow and, perhaps one day, watching you read these lines. May our Saviour guide your steps and continue to fill your heart with joy. Thanks for the ghostwriting (see Appendix E).

Last but certainly not least, Marie. You endured this PhD journey as much as I did, but without the praise or Thursday lunches. I am thankful for your daily expressions of love, your patience during countless evenings when I came home later than promised, your positivity in my moments of discouragement, your prayers when work tested my temper, and your endurance through endless working weekends and shortened vacations. These words can never compensate for the letters left unwritten or the sacrifices you made. Our wedding and Coline's birth have been the brightest highlights of these years. I look forward to our post-PhD life and to all the evenings I'll return home on time 🤞. Thank you for accepting this journey and for your unwavering support. May our love, rooted in Christ, continue to deepen and bring joy to our hearts.

Soli Deo Gloria

Lausanne, 7th April, 2024 Nicolas Badoux

Abstract

Software is pervasive in our daily lives. From remote-controlled traffic lights that orchestrate traffic flows to applications reminding us to take medications, it increasingly governs safety-critical systems. To ensure correct and continuous execution, bugs have to be detected before the software is rolled out in production. Low-level code—used in performance-critical or resource constrained environment—is often written in languages with limited security guarantees. Specifically, C and C++ lack type- and memory-safety, making them prone to bugs. As manual efforts do not scale to the size of contemporary codebases, this thesis addresses the challenge of securing system code through techniques requiring minimal manual effort. We tackle this challenge along three dimensions: a language-level protection against derived type confusions, a detector for *all* type confusions during testing, and lastly a novel approach to automate fuzzing of software libraries.

Our first contribution addresses illegal type conversions in C++ programs. Casting objects through an inheritance hierarchy, a common operation in C++, may lead to exploitable type confusion vulnerabilities. We propose type++, a novel C++ dialect that prohibits derived type confusions by design. By inlining type information in all objects involved in derived casts, type checks are performed at runtime for all casts, ensuring type safety. The change in object layout requires minimal patching and incurs minimal runtime overhead—less than 1% overhead on the SPEC CPU benchmarks, allowing type++ to discover 14 new vulnerabilities.

Our second contribution is Sourcerer, a comprehensive type confusion sanitizer. By instrumenting objects involved in unrelated casts on top of type++, Sourcerer detects all type confusions. Through a specialized type information initializer, RTTIInit, we reduce the porting effort and the performance impact. Sourcerer supports complex idioms such as unions, which were problematic for previous tools. With a performance overhead of only 5%, Sourcerer verifies twice as many unrelated casts as type++ on the SPEC CPU benchmarks. Notably, Sourcerer is the first sanitizer used for a fuzzing campaign specifically aimed at type confusions. Instrumenting OpenCV, we discover six new type confusions using seven fuzz drivers.

Lastly, we present LIBERATOR, a novel consumer-agnostic fuzz driver generator. Fuzzing relies on executing the code which is not directly possible for software libraries. Through static analysis and rapid feedback, LIBERATOR generates valid C drivers, exercising diverse and complex library functions. This addresses the limited development resources available for driver creation. Our evaluation shows that LIBERATOR achieves higher coverage in most libraries compared to the state-of-the-art and matches the coverage of manually written drivers. Notably, LIBERATOR discovers bugs in extensively tested libraries where other approaches fail. Finally, we characterize the trade-off between generating and testing drivers, a balance that LIBERATOR is first to fully explore.

This thesis demonstrates how novel approaches enhance the security of system code with minimal developer effort. By preventing most type confusions and detecting the rest, we offer a comprehensive solution to address the lack of type safety in C++. For C libraries, LIBERATOR

improves the testing workflow and scales to the numerous libraries underpinning today's software ecosystem.

Keywords: type confusion, type safety, fuzzing, library testing, C++

Résumé

Les logiciels sont omniprésents dans notre vie quotidienne, des feux de circulation télécommandés aux applications de semainier, et régissent de plus en plus de systèmes critiques. Pour assurer une exécution continue et correcte, les bugs doivent être détectés le plus tôt possible dans le processus de développement. Le code de bas niveau est souvent écrit dans des langages de programmation offrant de faibles garanties de sécurité. En particulier, le C et le C++ manquent de sûreté de mémoire et du système de type, les rendant sujets aux crashs. Les efforts manuels ne pouvant pas suivre la tailles des programmes actuels, cette thèse aborde le défi de sécuriser le code de bas niveau par des techniques d'automatisations.

Notre première contribution traite des conversions illégales en C++. La conversion d'objets, une opération courante en C++, peut conduire à des vulnérabilités de confusion de types. Nous proposons type++, un nouveau dialecte du C++ qui prévient les confusions de types dérivés. En intégrant le type dans les objets impliqués, la validité de la conversion peut être assurée durant l'exécution, assurant ainsi la fiabilité du système de types. Ce changement conduit à une nouvelle disposition en mémoire et nécessite une faible adaptation du code et engendre une perte de performance minimal tout en protégeant 90 milliards de conversions

Notre deuxième contribution, Sourcerer, est le premier détecteur complet de confusion de types. En instrumentant les objets impliqués dans les conversions sans relation, Sourcerer détecte toutes les confusions de types. Grâce à une nouvelle fonction, RTTIInit, nous réduisons à la fois l'effort d'adaptation au dialect et l'impact sur les performances. Sourcerer supporte des idiomes comme les unions, qui étaient problématiques pour les outils précédents. Malgré un surcoût en terme de performance de seulement 5%, Sourcerer vérifie deux fois plus de conversions sans relation que type++ sur les tests SPEC CPU2006 et CPU2017. De plus, Sourcerer est le premier détecteur utilisé durant une campagne de fuzzing ciblant spécifiquement les confusions de types. En instrumentant OpenCV, nous découvrons six confusions de types.

Finalement, nous présentons libErator, un générateur de harnais de test dynamique ne nécessitant pas de code tiers. Le fuzzing, en tant que technique de test dynamique, nécessite l'exécution du code de la bibliothèque. Grâce à une analyse statique et une évaluation rapide, libErator génère des harnais en C valides, exerçant des fonctions de bibliothèque diverses et de manière complexe. Ainsi, nous répondons au manque de harnais du à l'effort conséquent nécessaire à leur création. Notre évaluation montre que libErator exerce plus de code dans la plupart des bibliothèques par rapport aux outils concurrents et offre une couverture similaires aux harnais manuels. De plus, libErator découvre des bugs dans des bibliothèques extensivement testées. Enfin, nous caractérisons le compromis entre la génération et le test des pilotes, un équilibre que libErator est le premier à explorer.

Cette thèse démontre comment de nouvelles approches peuvent améliorer la sécurité du code de bas niveau avec un effort minimal du développeur. En prévenant la plupart des confusions de types et en détectant les restantes, nous offrons une solution complète pour répondre au manque

de sûreté des types en C++. Au vus du peu de harnais disponible pour fuzzer les bibliothèques écrites en C, libErator augmentent la pénétration des tests sans effort manuel.

Mot clés : confusion de type, sécurité de type, fuzzing, test de librairies, C++

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

> Safety does not happen by accident.
>
> *Author unknown*

"Software is eating the world," observed venture capitalist Marc Andreessen in 2011 [1]. This statement remains accurate as software continues to transform industries and daily activities—from physical newspapers to *drive-by-wire* vehicles. Increasingly, computers perform tasks where failures may have catastrophic consequences. For example, ensuring the correct and safe execution of software in flight management systems or pacemakers is paramount. Consequently, software failures can lead to substantial economic losses [25, 29] and even deaths [62, 73]. To mitigate these risks, security policies and techniques are gradually integrated into the software development process, such as security by design and software testing. When applied early in the development cycle, these approaches increase, indeed, the confidence in the software's safety and reliability. For example, a memory-safe programming language like Rust [85] eliminates many memory-safety issues by design. Interestingly, software rarely displaces itself. For instance, 5M LoC—12.5%—of the Linux kernel, which powers most of the internet, was written over 15 years ago, as shown in Figure 1.1. More importantly, less than two-thirds of the Linux code from 2010 has been replaced, indicating that many legacy implementations, written in older language specifications with lesser security guarantees, persist. Instead, software grows in size and complexity, surpassing the 100M LoC mark for projects like Chromium [20]. Such large-scale software projects face significant maintenance challenges. For instance, in the Linux kernel, each of the 68 core maintainers is responsible for over 100k LoC, highlighting the daunting task of ensuring code quality and security at scale. Additionally, today's software relies on a myriad of dependencies, simplifying development by reusing existing code as libraries, but also creating a single point of failure for the whole ecosystem. As our reliance on software increases, so do expectations for correctness and security. Newer code benefits from past lessons and employs modern paradigms with built-in safety, such as memory-safe programming languages. However, legacy code remains inherently insecure despite the advantage of time which eliminates low-hanging bugs. Moreover, migrating a codebase to a new language is a daunting task far from an automated process despite ongoing research, like DARPA TRACTOR [26] for migrating from C to Rust. This suggests that many projects will be stuck on outdated security paradigms.

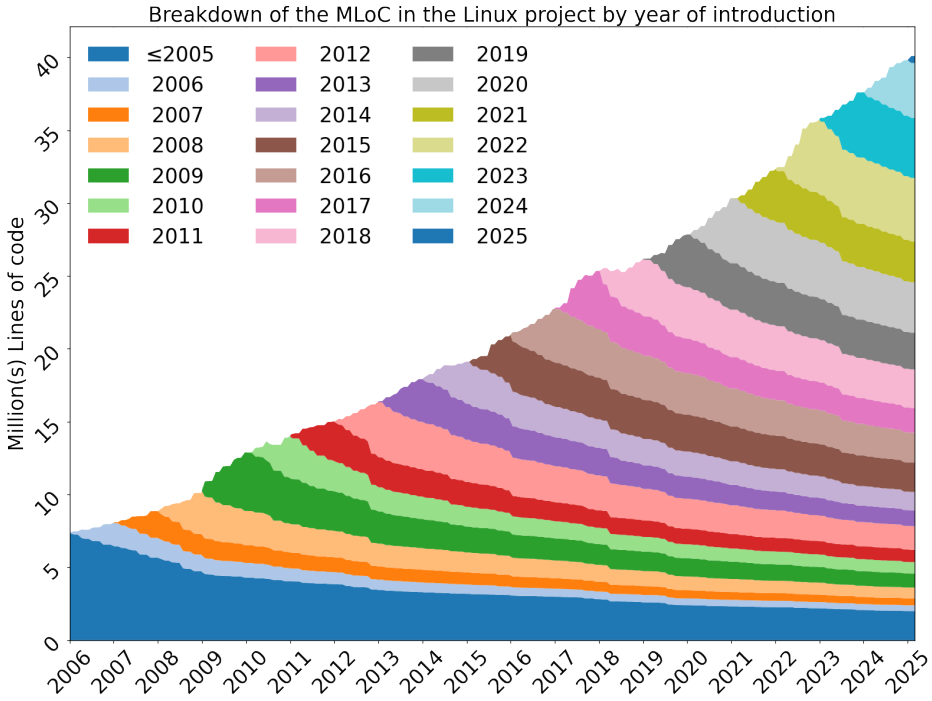

Figure 1.1 – Age of the lines of code in the Linux kernel. Legacy code rarely get replaced when new code is added, leading to the codebase growing in size and complexity. Blank lines and comments are included for technical reasons.

Vulnerabilities—software flaws that can be leveraged to compromise security—have many possible root causes, ranging from logic errors by the developer to out-of-date dependencies. While memory-safety issues have historically been reported as a significant portion of security bugs—with major software companies like Google [99], Microsoft [32], and Apple [64] attributing up to 70% [38] of all security bugs to such issues—recent efforts, in particular the adoption of memory-safe languages, have started to reduce their prevalence [**fewervuln**]. Nonetheless, securing code written in languages without a safe memory model and type system remains a critical challenge, particularly for legacy codebases, and motivates the work in this thesis.

More specifically, we focus on two programming languages, C and C++, which are the most widely used languages in low-level programming—ranking second and first, respectively, in popularity on GitHub in 2024 [40]. Due to the desire for retro-compatibility and their original design dating to the 1970s and 1980s, they lack a modern security focus, in particular in terms of memory-safety. C++, released as an extension of C, provides safer paradigms (*e.g.,* `string` instead of `char*` allowing for length checks) while offering similar performance to C. However, C++ still inherits inherent memory unsafety (*e.g.,* arrays carry their bounds neither in C nor in C++) and add complexity the language. Over the past decades, the C and C++ languages and their respective ecosystems have evolved to incorporate various safety improvements. Newer versions of

the language specifications introduced safer constructs like safer integer handling in C99 [102] or smart pointers in C++11 [4]. Moreover, the compilers have enhanced their analysis to provide accurate and precise warnings. Additionally, runtime protections like Address Space Layout Randomization (ASLR) [110], Data Execution Prevention (DEP) [124], and Control Flow Integrity (CFI) [14] have been introduced to mitigate exploitation. Finally, an emphasis on security testing early in the development cycle and the development of effective tools helped prevent many security issues. Specifically, the introduction of sanitizers—bug oracles that detect when an invariant is violated—in conjunction with dynamic testing techniques like fuzzing identified numerous bugs in C and C++ programs.

Memory safety violations often occur without visible program failures, making their detection particularly challenging. Modern sanitizers address this challenge by instrumenting programs to detect safety violations at runtime, trading performance for comprehensive detection capabilities [111]. For instance, AddressSanitizer [106] detects out-of-bounds accesses by placing tripwires, special memory regions (*red zones*) around allocated objects, incurring a 70% performance overhead but providing strong spatial safety guarantees during testing.

In addition to logic bugs possible in any programming language, C and C++ are prone to memory safety violations, which can be categorized into five fundamental classes [100]:

- **Spatial safety violations:** Each allocated object occupies a specific memory region with defined boundaries. Any access outside these boundaries (*e.g.,* reading beyond an array's allocated size or dereferencing a pointer that has been incremented past its valid range) constitutes a spatial safety violation.
- **Temporal safety violations:** Similarly to spatial safety, every allocated object has a defined lifetime, from allocation to deallocation. Accessing an object after its lifetime has ended (e.g., dereferencing a dangling pointer after `free()` or returning a pointer to a stack-allocated variable) creates a temporal safety violation, commonly known as use-after-free and use-after-return, respectively.
- **Initialization safety violations:** Using memory that has been allocated but not explicitly initialized leads to undefined behavior. Such accesses may expose residual data from previous allocations, potentially causing information leaks. Moreover, using uninitialized values as pointers or array indices can trigger additional safety violations.
- **Type safety violations:** Converting between object types allow for flexible code and is well-defined for specific type pairs. Interpreting memory using an incompatible type, however, may lead to incoherent values or further spatial safety violations. Such vulnerabilities are referred to as type confusions.
- **Thread safety violations:** In concurrent programs, shared memory access must be properly synchronized between threads. Without adequate synchronization primitives (*e.g.,* mutexes,

atomic operations), simultaneous access to shared data can result in race conditions, corrupted memory state, or inconsistent program behavior.

The landscape of sanitizers has evolved to address specific categories of memory safety violations, each offering distinct detection capabilities and performance trade-offs. For instance, AddressSanitizer (ASan) [106] identifies spatial and temporal safety violations, while ThreadSanitizer (TSan) [107] and MemorySanitizer (MSan) [112] focus on thread and initialization safety violations, respectively. Beyond these widely adopted tools, researchers have developed specialized solutions such as SoftBound [91] and CETS [90], which provide comprehensive guarantees for spatial and temporal safety. However, type safety violations remain particularly challenging to detect comprehensively. While C's weak type system inherently permits, and sometimes encourages, type violations, C++ offers stronger typing mechanisms that facilitate both detection and prevention of type safety violations. Consequently, existing type safety sanitizers [49, 59, 70] target C++ but currently protect only a subset of objects, falling short of complete type confusion detection coverage.

To maximize the benefits of sanitizers, which enhance the bug-detection sensitivity of a testing session, it is crucial to address other dimensions of software testing. In particular, expanding the extent of code tested is essential. While sanitizers effectively detect violations during runtime, their utility is inherently limited by the scope of the code exercised during testing. While manual testing efforts are valuable, they are inherently limited by the maintainers' time and resources. Though automated test generation techniques exist [34, 125], industry [109] has widely adopted fuzzing as the primary automated testing approach. Modern fuzzing frameworks [33, 133, 140] leverage significant computational resources to systematically explore program behavior by generating and executing millions of test cases within hours. These frameworks use code-coverage feedback from program execution to guide input generation toward unexplored paths [133] or potentially vulnerable code regions [11]. While fuzzing has proven effective across various domains, from kernel [103] to trusted execution environments [15] testing, fuzzing software libraries presents unique challenges. Libraries cannot be directly executed; instead, they require a driver program that exercises their Application Programming Interface (API). Creating effective drivers demands deep understanding of the library's architecture, expected usage patterns, and internal state management. This expertise requirement, combined with other development priorities, often results in a scarcity of fuzz drivers, limiting the effectiveness of library testing efforts.

The two aforementioned challenges—type confusion in C++ and library fuzzing—form the backbone of this thesis and lead to the following research questions:

Research questions

How can we enhance the security of the swath of legacy code written in memory-unsafe languages by:

1. Protecting against derived type confusion vulnerabilities in C++ at runtime,
2. Efficiently detecting *all* type safety violations in C++ during testing, and
3. Automating testing to find vulnerabilities in C libraries?

This thesis addresses these questions through three core contributions:

(i) a novel mitigation, type++, targeting the most frequent type conversions in C++ programs,

(ii) a complete sanitizer, Sourcerer, to detect type confusions in C++ programs, and

(iii) a study of the challenges inherent to automated library fuzzing and a framework, LIBERATOR, addressing them.

Before detailing these contributions and formulating the thesis statement in §1.3, we briefly introduce the underlying issues and motivate our contributions.

1.1 Type confusion in C++

Type conversion is a fundamental operation in C++, enabling flexible and generic programming patterns. However, unlike type-safe languages, C++ permits unsafe type conversions that can compromise memory safety. For instance, `reinterpret_cast` allows developers to convert between arbitrary types, regardless of compatibility of the underlying memory layout, resulting in potential out-of-bound access. While the C++ standard clearly defines which conversions are safe and legal, developers often perform unsafe conversions due to C programming habits, insufficient understanding of type safety, or implementation oversights. These unsafe conversions can lead to type confusion vulnerabilities (tracked under CWE-704 and CWE-843), which have been repeatably discovered in major C++ projects including Firefox (CVE-2023-25736, CVE-2023-25737) and Google Chromium (CVE-2019-5757, CVE-2020-6464, CVE-2022-3315, CVE-2023-6348).

1.1.1 Derived type confusion

In object-oriented C++, inheritance and polymorphism allow child classes to be used in place of their parent classes. The reverse operation—using a parent object where a child type is expected—requires an explicit cast operation known as *derived cast* or downcast. Such casts are inherently

unsafe: for a derived cast to succeed, the object must have been allocated as the derived type or one of its descendants. Otherwise, the cast might allow access to out-of-bounds memory since child types are equal or larger than their parents.

Statically verifying derived cast safety is intractable due to type aliasing introduced by polymorphism and generic pointer types like `void*`. While runtime checks offer the only viable solution, C++ objects lack the necessary type information for runtime checks by default. For polymorphic types (the subset of classes with at least one virtual function), C++ provides `dynamic_cast` which performs runtime type checking. However, when either the source or destination types are non-polymorphic, the C++ standard mandates the use of unsafe cast operators like `static_cast` or `reinterpret_cast`, shifting the burden of ensuring type safety to developers. Misuse of these operators leads to derived type confusions (type confusions), resulting in undefined behavior.

Preventing and detecting type confusion bugs has been an active research area over the last decade. CaVer [70] pioneered runtime checking of all derived cast operations. Subsequent works like TypeSan [49] and HexType [59] improved performance through optimized data structures for tracking object metadata throughout their lifetime. In parallel, the LLVM project—a compiler toolchain—developed a defense for objects with RTTI. Their type checking scheme automatically lifts unsafe casts to `dynamic_cast` operations, ensuring type safety through runtime checks. While these checks, available through LLVM-CFI [120], incur minimal overhead ($<$1%), they only protect polymorphic objects. CastSan [89] further improved this approach by optimizing class hierarchy membership analysis, achieving even faster runtime checks.

These two approaches represent the current state-of-the-art in type confusion prevention and detection: sanitizers providing protection for all derived casts at high runtime cost, and compiler-based solutions offering efficient but partial protection. This leaves an important gap: protecting all derived casts from type confusion while maintaining practical performance overhead. To address this challenge, in Chapter 2, we present type++, a novel C++ dialect allowing for inline type metadata and fast type checks. Once ported to this new dialect, a program is free from derived type confusions. The instrumentation incurs a minimal runtime overhead, on average less than 1%, allowing for its deployment as a mitigation.

1.1.2 Unrelated type confusion

C++ supports type conversions beyond inheritance hierarchies. While some conversions are well-defined—like user-defined conversions (`To(From&)`) or standard implicit conversions (*e.g.,* `bool` to `int`)—others, called *unrelated casts*, can lead to undefined behavior. These unrelated casts fall into two categories: First, conversions involving integral types and pointers—*e.g.,* voidp. Any pointer can be safely converted to a sufficiently large integral type (*e.g.,* `uintptr_t` or `void*`). The reverse conversion is only defined when the memory contains an object of the target type.

For example, converting a pointer to `void*` and back to its original type is safe, but converting to any other type results in undefined behavior. Second, conversions between unrelated types lacking developer-defined conversion paths. These casts always result in undefined behavior. However, some programs still rely on such conversions due to C++ compilers' conservative object layouts.

Current sanitizers primarily focus on derived casts, offering limited protection for unrelated casts. LLVM-CFI [120] only verifies casts to polymorphic types and may miss violations in adversarial scenarios. While Ironclad-C++ [27], a C++ dialect, bans unsafe casts entirely, only EffectiveSan [30] attempts to detect unrelated casts through runtime type checking, albeit with significant overhead. In Chapter 3, we present Sourcerer, a novel sanitizer designed to detect all type confusions efficiently. Building on type++, Sourcerer extends detection to unrelated casts and unions through a new type information initializer, RTTIInit. Our approach achieves comprehensive type checking with only 5% overhead on the SPEC CPU benchmarks. Moreover, Sourcerer enables the first successful fuzzing campaign targeting type confusions, discovering six new vulnerabilities in OpenCV using AFL++ [33].

1.2 Library fuzzing

Testing software libraries presents unique challenges compared to standalone applications. To achieve meaningful coverage, library testing requires exercising diverse API functions while building coherent program states. To execute the library's code, specialized programs, called fuzz drivers, are written. They invoke library functions in valid sequences with appropriate arguments, mimicking minimal consumer programs. Creating effective drivers demands deep understanding of: (i) the API's function ordering constraints (*e.g.,* initialization, cleanup), and (ii) relationships between an API function arguments (*e.g.,* buffer and its size). Due to these knowledge requirements, fuzz drivers are typically written by library maintainers, but are usually few and far between due to the competing development priorities with new features often receiving higher priority than testing or even maintaining existing code. This scarcity of drivers limits their effectiveness as they cover a reduced number of API functions. Therefore, automating driver creation unleashes fuzzing's unmatched bug-finding capabilities on the vast ecosystem of C/C++ libraries. This requires, however, understanding complex API dependencies to generate valid usage patterns.

This has been the focus of different work through two main approaches. First, FuzzGen [56], Fudge [5], OSS-Fuzz-Gen [54], and UTopia [60] learn API usage from existing code or test cases. While effective, these consumer-dependent approaches cannot exercise functionality absent from their training data. On the other hand, Hopper [18] takes a consumer-agnostic driver generation approach by fuzzing libraries through an interpreted intermediate representation. However, this approach limits fuzzer integration, constraining its effectiveness. The challenge of generating diverse and valid drivers at scale remains. In Chapter 4, we present libErator, a novel

consumer-agnostic approach that generates valid C drivers compatible with any fuzzer. Through static analysis and rapid testing feedback, libErator creates drivers that exercise complex library functionality. More importantly, libErator characterizes the trade-off between driver generation and testing time, investing a problem faced but not addressed by previous tools. Our evaluation shows that libErator finds bugs in well-tested libraries where other approaches fail, while achieving comparable or better coverage than existing techniques.

1.3 Thesis contribution

This thesis advances the security of C and C++ code through novel approaches that enhance both software testing effectiveness and runtime safety. We focus on two critical challenges: (i) preventing and detecting type confusion vulnerabilities in C++, (ii) and automating the testing to software libraries. Towards these goals, we make three core contributions:

(i) type++: a novel C++ dialect that prevents derived type confusion by design with minimal runtime overhead, enabling its use as a mitigation in production;

(ii) Sourcerer: a comprehensive sanitizer that efficiently detects all type confusion vulnerabilities during testing; and

(iii) libErator: an automated approach for generating fuzz drivers that enables balancing between shallow and deep testing of C libraries without requiring existing consumer code.

Together, these contributions demonstrate how language-level changes and automated techniques can significantly improve software security while minimizing developer effort.

1.3.1 type++

type++ is a novel C++ dialect that prohibits derived type confusion by design. By augmenting objects involved in derived casts with inline Runtime Type Information (RTTI), type++ can perform runtime checks throughout object lifetimes, guaranteeing type safety. Our instrumentation requires minimal code changes, typically affecting less than 0.04% of the codebase. Inlining type metadata avoids the expensive object tracking of previous sanitizers, requiring instrumentation only at cast sites and object allocations. Implemented on top of LLVM, type++ provides warnings for incompatible idioms to assist developers during porting. The protection added by type++ incurs negligible runtime cost, allowing its deployment as a mitigation. Additionally, type++ is free of false positives by design while providing complete protection for derived cast operations.

Our evaluation using SPEC CPU2006 and CPU2017 benchmarks demonstrates type++'s compatibility with the C++ standard, minimal runtime cost (<1%), and effectiveness at preventing derived type confusions. Our results show protection for up to 23× more casts than previous mitigations while identifying 122 type confusions, including 14 previously undetected issues. A case study on Chromium further shows that partial deployment of type++ is practical, protecting 94.6% of vulnerable classes with less than 1% runtime overhead.

1.3.2 Sourcerer

The second core contribution of this thesis is Sourcerer, the first comprehensive type confusion sanitizer with practical runtime overhead. While type++ prevents derived type confusion, Sourcerer extends detection capability to all type conversions, including unrelated casts. As the first tool to reliably check casts from generic pointers (*e.g.,* `void*`), Sourcerer instruments all classes involved in explicit cast operations. To facilitate the necessary change, Sourcerer introduces a specialized type information initializer, RTTIInit, that reduces the divergence between our dialect and standard C++. This approach achieves complete type safety—even for unions that remained so far unsupported—while maintaining a modest runtime overhead of 5.14% on the SPEC CPU benchmarks. Our evaluation shows that Sourcerer protects twice as many unrelated casts as type++ and detects 30 previously unknown type confusions in the SPEC CPU benchmarks. More significantly, we conduct the first successful fuzzing campaign specifically targeting type confusion vulnerabilities. By instrumenting OpenCV, a widely-used computer vision library, with Sourcerer and AFL++, we discover six new type confusion vulnerabilities, demonstrating the effectiveness of combining comprehensive type checking with modern fuzzing techniques.

1.3.3 libErator

Our third contribution, libErator, addresses the challenge of automated library testing through a novel driver generation technique that operates without requiring consumer code. We first characterize the fundamental trade-off between time spent generating drivers and time spent testing them—a challenge faced but not addressed by previous work. Based on this insight, libErator generates fully-featured C drivers while allowing developers to optimize this trade-off for their specific libraries. libErator employs static analysis to infer both argument dependencies within API functions and relationships across function calls. From this analysis, it constructs API chains—sequences of function calls that maintain program state consistency. Each chain is augmented with necessary setup and cleanup code. To avoid wasting testing resources, libErator uses rapid testing feedback to eliminate redundant or ineffective chains early in the process.

Our evaluation demonstrates libErator's effectiveness in finding bugs even in well-tested libraries. Compared to consumer-aware approaches, libErator achieves similar or better cover-

age. More importantly, when compared to competitors, both consumer-agnostic and consumer-dependent tools, LIBERATOR explores more or similar amount of code paths in the 15 target libraries and discovers 24 new bugs—where previous tools found at most two.

1.4 Thesis Statement and Organization

Thesis statement

By incorporating safety at the language-level and through automation, system code can be protected with minimal human involvement. First, through a novel dialect of C++, we enable type safety for large code bases. Second, by automating fuzz driver creation, we scale testing to low-level library APIs.

Thesis Organization. This thesis is distributed across three chapters. Chapter 2 describes the type++, a novel C++ dialect free of derived type confusion. Chapter 3 details Sourcerer, the first complete type confusion sanitizer capable of fuzz testing. Finally, Chapter 4 presents LIBERATOR, a fuzz driver generation technique balancing generation and testing when fuzzing library.

Bibliography Notes. This thesis was supervised by my advisor, Prof. Mathias Payer. The sections of this thesis describe projects conducted in collaboration with academic peers, namely Flavio Toffalini, Yuseok Jeon, and Zurab Tsinadze. This thesis contains contributions from the following conference publications:

- Nicolas Badoux et al. «type++: Prohibiting Type Confusion With Inline Type Information». In: (Feb. 2025). DOI: 10.14722/ndss.2025.23053. URL: https://dx.doi.org/10.14722/ndss.2025.23053

  which received the *Distinguished Paper Award*.

- Nicolas Badoux, Flavio Toffalini, and Mathias Payer. «Sourcerer: channeling the void». In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2025

- Flavio Toffalini et al. «Liberating Libraries through Automated Fuzz Driver Generation». In: *Proceedings of the ACM on Software Engineering,* ACM New York, NY, USA, July 2025. DOI: 10.1145/3729365. URL: https://dx.doi.org/10.14722/3729365

Doctoral candidate's contributions.

- **type++.** The doctoral candidate is the first author and the main contributor to the paper. The doctoral candidate designed the type++ dialect, implemented the compiler, and conducted three quarters of the evaluation.
- **Sourcerer.** As the first author and main contributor to the paper, the doctoral candidate implemented the compiler, and conducted the evaluation.
- **libErator.** The doctoral candidate contributed to the design throughout the project, some implementation, and conducted a third of the evaluation. He contributed extensively to the writing of the paper.

Chapter 2

type🇨🇭🇨🇭: Prohibiting Type Confusion With Inline Type Information

J'ai compris que ça parlait de fantômes 👻

My mum

The C++ language provides high-performance and object-oriented abstraction capabilities. C++ organizes objects in parent-child relationships, in which child classes inherit (and extend) attributes and functions from their parents. Ideally, any new class represents a new type in the program, thus designing flexible software architectures. Casting operations enable more modular software development as objects can be passed to functions defined for another type. Unfortunately, C++ does not enforce type safety and cannot check the correctness of all cast operations due to limited type information available at runtime, resulting in the risk of type confusion. Type confusion happens when the program interprets an object as belonging to a different type. In these attacks, fields may be interpreted as different types (*e.g.,* an `int` misinterpreted as a pointer) or invoking unexpected virtual functions. Type confusion vulnerabilities in C++ applications are a building block to mount code-reuse [104] and data-only attacks [57] in a wide range of software products, such as Telegram (CVE-2021-31318), Firefox (CVE-2023-25736, CVE-2023-25737), or Google Chromium (CVE-2019-5757, CVE-2020-6464, CVE-2022-3315, CVE-2023-6348). Across all these CVEs, developers tried to detect these vulnerabilities (*e.g.,* using `static_cast` or custom runtime type information) but ultimately failed due to mismatches between the expected static types in the source code and the dynamic type of the objects at runtime. Type confusions are reported under the Common Weakness Enumeration *CWE-704: Incorrect Type Conversion or Cast* and caused by *CWE-843: Access of Resource Using Incompatible Type*.

Considering the impact of type confusion attacks and other memory safety issues, researchers designed new languages that are type-safe by design, such as Java or Rust [85]. These languages prohibit erroneous typecasts, mitigating this attack surface. As C++ remains widely used in legacy C++ codebases, protecting C++ from type-safety violations is crucial. However, due to its compatibility with C and the lack of inherent runtime type information for non-polymorphic objects, enforcing type safety in C++ presents significant challenges. This compatibility introduces a duality between C++ classes and C-style structs, where once an object is allocated, its type

cannot inherently be retrieved from (raw) memory. While polymorphic objects benefit from runtime type information through virtual table pointers (*vptr*), non-polymorphic objects remain opaque at runtime, leaving them vulnerable to type confusion attacks. due to the compatibility between C++ and C, enforcing type safety and thereby mitigating type confusion is challenging. For interoperability, C++ provides an almost duality between classes and C-style structs. Once an object has been allocated, there is no inherent way of retrieving its type from the memory encoding. In C, casts merely determine the fields' offset. With polymorphic objects (those with virtual functions), C++ introduces vtable pointers (*vptr*), *i.e.,* Runtime Type Information (RTTI) to uniquely identify the type of the object. Without vtable pointer, the type of a memory area remains opaque at runtime. C++ provides different cast operations offering varying guarantees. Only dynamic casts execute a runtime check, but can only be used for polymorphic objects since they rely on the RTTI of vtable pointers. Conversely, static casts leverage the declared types in the source code to statically check class relationships while C-style casts carry neither a dynamic nor static check. To enforce type safety, only `dynamic_cast` should be used [117], an option that core contributors of the standard envision to ease even at the cost of breaking the Application Binary Interface (ABI) [116].

LLVM-CFI [120] leverages RTTI to protect casts involving the fraction of polymorphic objects, *i.e.,* only around 3% of all the casts in our benchmarks. Alternatively, the academic community proposed several sanitizers [30, 49, 59, 69, 80, 81, 95] that track all objects involved in down-to casts and check their types against disjoint metadata. However, they incur high overhead and are imprecise (*e.g.,* if object copy is incorrectly tracked), resulting in both false positives and false negatives. Therefore, C++ remains prone to type confusion, a common attack vector used in exploits.

By introducing type++, a C++ dialect, we tackle type confusion vulnerabilities from a language perspective. By relinquishing some compatibility (with negligible impact in practice), type++ checks down-to casts at runtime, guaranteeing that objects are only used under the respective types specified at initialization. Our C++ dialect builds on the property that *each object is typable throughout its lifetime*. type++ associates a unique type (*i.e.,* a hidden field) to each object. This inline type field is key for fast and effective type checks for all casts, similar to other type-safe languages (*e.g.,* Java or Rust). At the same time, type++ is highly compatible with existing C++ programs. The few incompatible code patterns are easy to detect, and therefore amenable to straightforward patching. We analyze and classify the code adaptations that type++ requires. For each, we study solutions and measure the impact of such modifications on real cases. Where the code patterns are incompatible with type++, we devise compile-time warnings and errors that aid developers in porting code to the type++ dialect.

To demonstrate the efficacy of our dialect, we implement type++ by extending Clang and LLVM. Our compiler takes C++ programs as input, produces warnings in case of incompatibilities,

and then emits executables protected against downcasting type confusion.[1] We deploy type++ across the two compiler benchmarks SPEC CPU2006 [50] and SPEC CPU2017 [13]. Additionally, we develop a benchmark to compare type++ to external metadata approaches. Moreover, we apply our type++ compiler to Chromium to show compatibility with, likely, today's most complex C++ code base. We choose the SPEC CPU benchmarks because their use cases range from simple examples (*e.g.,* NAMD ~3 kLoC) to large, complex programs (*e.g.,* Blender ~600 kLoC) and because it allows comparison with previous works. We successfully compile around 2,040 kLoC, encompassing both SPEC CPU2006 and CPU2017. Chromium is a massive, fast-moving project, and only setting up the environment to compile it with a different LLVM version may take months. We manage to protect 3,030 classes (out of 3,201 — 94.6%) in roughly twelve man months, while we deem the remaining classes easy to address given developers familiar with the codebase. For all our use cases, we measure the performance overhead, the number of lines of code modified during the porting, and compare it with existing runtime type confusion mitigations.

Our results show that type++ only incurs on average a 0.94% overhead when considering both SPEC CPU2006 and CPU2017 together, which is two orders of magnitude faster than HexType, a state-of-the-art type confusion sanitizer. Additionally, type++ validates 23 times more casts than LLVM-CFI. Compared to previous approaches, type++ does not suffer from false-positive type confusion detection since it is free from incorrect object lifetime tracking that plagued the previous type confusion detectors. Finally, type++ remains highly compatible with the current C++ ISO Standard. When looking at the SPEC CPU benchmark suite, inspecting the warnings resulted in modifying 125 LoC and 131 LoC for SPEC CPU2006 and CPU2017, respectively. For Chromium, our porting modifies only 229 LoC out of the 35 MLoCthe project contains.

To sum up, our contributions are:

- Specification of type++, a C++ dialect that enforces safe down-to cast by design through our baseline property that all objects remain typable throughout their lifetime.
- Study the adaptation required to port standard C++ programs over to type++ and propose an analysis/mechanism to help developers with this task.
- Evaluation of type++ against both SPEC CPU2006 and SPEC CPU2017, demonstrating its high compatibility and low performance overhead.
- Characterization of the porting for Chromium on type++ through a twelve months long study, along with a discussion of this effort.

The full source code, the documentation to replicate our experiments, and our technical reports are open-source: `https://github.com/HexHive/typepp/`.

[1] Unless differently specified, we use the term type confusion to refer to downcasting type confusion throughout the chapter.

2.1 Background

C++ is an object-oriented programming language with diverse features and constraints due to the tight relationship with the systems programming language C. Here, we detail the concept of class hierarchies (§2.1.1), C++ casting operations (§2.1.2), and the type confusion problem (§2.1.3).

2.1.1 Classes hierarchies and polymorphism

C++ relies on classes to implement polymorphism, which allows a developer to abstract generic concepts and reduce code duplication in a program. In the simplest schema, classes are organized hierarchically in parent-child relationships. This approach allows child classes to extend or override some functionality of the parent. In particular, C++ implements function override using a vtable, which is a table of function pointers to so-called virtual functions [113]. For instance, the classes `Greeter` and `Execute` in Listing 2.1 are both children of the class `Base`. Conversely, the class `UnrelatedGreeter` is disconnected from the inheritance tree. Defining a new class means introducing a new type in the program according to the language's type system. Therefore, an object of type `Greeter` cannot be used where an `Execute` or `UnrelatedGreeter` object is expected.

2.1.2 Casting in C++

Casting allows the programmer to reinterpret the runtime type of an object. In addition to implicit conversion, C++ features five distinct casting operators with distinct rules and properties. The developer is responsible for choosing the appropriate casting operator.

- `const_cast` strips the `const` and `volatile` property of an object. This operation has its caveats but is unrelated to type confusion. They are thus considered out of scope.

- `dynamic_cast` changes the type of an object with runtime verification validating the compatibility of the object's type. The validation routine mandated by the language specification generally requires the presence of metadata in the form of Runtime Type Information (RTTI). As RTTI is only available for polymorphic types, `dynamic_cast` cannot be used for non-polymorphic objects. The C++ Core Guidelines recommends `dynamic_cast` in almost all use cases as the performance impact is, despite rumors, negligible [117].

- `static_cast` changes the type of an object similarly to `dynamic_cast`. However, no runtime check is performed nor is RTTI required. Limited casting validation is performed at compile time, namely the check searches for a path in the type hierarchy from the source to the target type. Lacking any runtime verification, wrong casts (*e.g.,* an object of type `Parent` could be cast into `Child`) lead to type confusion vulnerabilities. Casting a pointer from outside the class

hierarchy (*e.g.,* `char*` or `void*`) into an object of a specific class is permissible as well. Incorrect usage of `static_cast` is the main culprit for type confusion in C++ applications.

- The `reinterpret_cast` operator reinterprets the underlying memory area of the object as the target type. Verification happens neither at compile time nor at runtime, and the correctness is delegated to the programmer, thus raising similar security issues to `static_cast`.

- In C++, C-style casts are automatically replaced at compilation time by following a priority schema: a `const_cast` is preferred over a `static_cast`, which in turn is favored over a `reinterpret_cast`. The first operator that satisfies compilation is selected. Due to their definition in terms of the other C++ casting operators, C-style casts inherit the underlying potential for type confusion vulnerabilities.

2.1.3 Type confusion

Type confusion vulnerabilities fall into two categories, namely *down-to-cast* and *unrelated type* (or `void*`) casting. *Down-to-cast* are portrayed in the functions `downToCast()` (line 28) while an *unrelated type* cast occurs in the `unrelatedType()` function (line 34) in Listing 2.1.

Down-to-cast vulnerabilities happen when a base class is cast into a subclass. In our example, the program treats an instance of a class `Execute` as if it is an instance of the `Base` class (line 42). Then, the base class is passed to the function `downToCast()` (line 43) which finally casts it into a `Greeter` object (line 29). At this point, the program erroneously considers the pointer `g` as a `Greeter`, however, the entry in the vtable of `g` points to the function `exec` that activates the payload (line 31) instead of the harmless `sayHi` function.

Unrelated cast vulnerabilities, instead, exploit backward compatibility for C programs that allows pointers to be considered as a generic type `void*`. In our example, the pointer `b` is an instance of class `Execute` (line 42). When `unrelatedType()` takes `b` as input (line 44), the compiler loses any type information of `b`. Therefore, the pointer can be cast as `UnrelatedGreeter` (line 35). The wrong cast allows an adversary to invoke `exec()` (line 37).

In the literature, there are two main approaches to mitigate these issues. First, approaches based on disjoint metadata structures trace and check the type of objects at runtime for down-to-cast [49, 59, 71, 95]. However, these approaches introduce considerable overhead and low precision. For instance, incorrect tracking of the object lifecycle (*e.g.,* due to not propagating metadata during object copies) causes stale metadata which might result in both false positives and negatives. False positives arise when the allocator reuses the space of a previously deallocated object without updating the metadata, thus leading to a misalignment between the actual object type and the stale metadata. False negatives, instead, appear when an object's metadata is unavailable and the system cannot decide whether a type violation occurred. Second, approaches may use existing (partial) type information to implement (partial) runtime type checks, *e.g.,* based

```cpp
#include <iostream>

class Base { // Parent class
  /* Other fields and functions */
};

class Execute : public Base { // Child of Base
public:
  virtual void exec(const char *program) {
    system(program);
  }
};

class Greeter : public Base { // 2nd Child of Base
public:
  virtual void sayHi(const char *str) {
    std::cout << str << std::endl;
  }
};

class UnrelatedGreeter { // Unrelated class
public:
  virtual void sayHello(const char *msg) {
    std::cout << "Hello: " << msg << std::endl;
  }
};

void downToCast(Base* b, const char *msg) {
  Greeter *g = static_cast<Greeter*>(b);
  // exec() is invoked instead!
  g->sayHi(msg);
};

void unrelatedType(void* p, const char *msg){
  UnrelatedGreeter *g = reinterpret_cast<UnrelatedGreeter*>(p);
  // exec() is invoked instead!
  g->sayHello(msg);
};

int main(int argc, char *argv[]) {
  const char *payload = "/bin/bash";
  Base *b = new Execute();
  downToCast(b, payload);
  unrelatedType((void*)b, payload);
  delete b;
  return 0;
}
```

Listing 2.1 – Example of a C++ class hierarchy. The functions `downToCast` and `unrelatedType` are examples of *down-to-cast* and *unrelated cast* type confusion vulnerabilities, respectively.

on pre-existing RTTI metadata and `dynamic_cast` in C++. Unfortunately, C++ allocates RTTI

metadata only for polymorphic objects, thus limiting the type check coverage. type++ overcomes the limitations of both approaches by defining a new C++ dialect that allocates inline metadata for all objects involved in downcasting, enforcing precise type checks, thus prohibiting type confusion by design.

2.2 Threat Model

As type++ can be deployed both as a sanitizer and as a mitigation, our threat model encompasses both scenarios.

We assume an adversary with knowledge of an illegal downcast, *i.e.,* the adversary knows where in the source code the downcast is and can construct an input to trigger a type confusion through this downcast. Our work prevents such type violations. More precisely, and in line with previous work [49, 59], arbitrary writes are out of scope for our mitigation. We assume that the attacker cannot modify the type information embedded in objects. This assumption is challenging to guarantee in practice but deploying mitigations like DEP, stack canaries, ASLR, CFI, and safe allocators [9, 84] approximates it. However, even with such mitigations, all out-of-bounds writes may not be prevented. For example, given a `strcpy` missing a length check and an array of objects whose layout ends with a `char*` object, an out-of-bound write may overwrite the next object vtable pointer.

Additionally, we assume a correct implementation of the type++ compiler and of the underlying operating system. Finally, as type++ does not rely on secrets, arbitrary reads do not generally compromise type++'s guarantees [84]. Leaking code pointers may allow an attacker to circumvent ASLR. While there are already many code pointers (*e.g.,* vtable pointers) available as target, type++ will increase the number of code pointers and, thereby, augment this attack surface.

2.3 type++ specification

C++ cannot enforce type safety due to the compatibility requirements of C++ objects with C structs. Also, historically, C++ compilers ran into performance and optimization constraints for type checks [113]. Yet, type violations are frequently used as the initial bug in exploit chains. Enforcing type checks as part of a safe C++ dialect mitigates these security risks. Essentially, C++ strictly associates a type only to polymorphic classes, all other structures and classes remain vulnerable to type confusion bugs. For simplicity's sake, we refer to classes and structures interchangeably. Our proposal, type++, is a novel C++ dialect that mitigates downcasting type confusion by design. In particular, type++ assigns a type to all objects of a program and en-

forces runtime checks for all down-to casts. We call this concept *Explicit Runtime Types*, formally described as:

Property 1 (Explicit Runtime Types.) *Given all classes CS of a program P, type++ associates a unique type T to each class $A \in CS$ at compile time. Embedding the type T into each object of class A enables explicit type checks before each downcast, ensuring their correctness.*

In practical terms, Property 1 ensures that each instance of a class A embeds a field that uniquely identifies its type. In classic C++, instances of polymorphic classes embed a vtable pointer for this purpose and a pointer in the RTTI section encodes the type information. Our property generalizes this requirement from polymorphic objects to all objects. We enforce this property at compile time. The introduction of this extra pointer allows a program to implement strict downcasting protections, thus mitigating completely these type confusion attacks. In our evaluation (§2.5), we show that adopting type++ provides strong protection at low performance overhead. Our system can, therefore, be used both as an effective sanitizer (helping developers discover and fix bugs) and as a powerful mitigation (preventing exploitation of type confusion vulnerabilities through early program termination).

Enforcing Property 1 protects all objects, thus achieving strong type safety mitigation by design. In practice, however, type confusion attacks require the execution of a cast. Thus, adversaries target only classes involved in these casts. Therefore, without loss of generality or security, adding type information only to *classes involved in downcasting operations* achieves the same security guarantees as protecting all classes. Furthermore, blindly embedding type information into all classes might change assumptions in the code about a class, thus possibly reducing the compatibility between type++ and the C++ standard. In type++, we consider that a class is involved in downcasting operations if anywhere in the program an object of that class is either cast to a different type in the same hierarchy or if a cast returns an object of this type. Upon these considerations, we introduce a additional property, called *Explicit Runtime Types for Cast Objects*, that specializes Property 1 and assigns a runtime type only to those classes at risk of down-to type confusion attacks. We formalize this property as follows:

Property 2 (Explicit Runtime Types for Cast Objects.) *Given the classes CS of a program P, type++ associates a unique type T to each class $A \in CS$ if A is involved in a downcasting operation.*

To enforce Property 2, we infer, during a compile-time analysis phase, which classes are involved in downcasting. Then, we associate a unique type T only to those classes involved in casting operations. In §2.5, we measure the impact of this optimization and show that enforcing Property 2 on SPEC CPU required us to change only 314 LoC out of 2,040 kLoC (around 0.04%).

To further improve the compatibility between standard C++ and type++, a developer can restrict Property 2 only to a subset of classes, manually annotated in the source code. This allows

developers to apply type++ only to some program components, thus reducing the number of incompatibilities to address. This feature is implemented as an additional property called *Explicit Runtime Types for Annotated Objects*, formally described as:

Property 3 (Explicit Runtime Types for Ann. Objects.) *Given the classes CS of a program P, and a set of annotated classes $AC \subseteq CS$, type++ associates a unique type T to each class $A \in AC$.*

The enforcement of Property 3 enables us to deploy type++ on complex programs, such as Chromium, by using limited resources (i.e., graduate students unfamiliar with the Chromium code base) and protecting 94.6% of its classes. Further information about this case study is provided in §2.5.4.

In the rest of this section, we discuss the differences between type++ and standard C++. Without loss of generality, we mainly consider Property 1 as it has stricter assumptions, while we switch to Property 2 or Property 3 only in clearly stated specific cases.

2.3.1 Affected programming patterns

type++ relies on Property 1 to ensure strong typing. However, this protection also introduces new assumptions not considered in the C++ standard and that lead to different programming patterns. Here we discuss the new patterns introduced by type++ and how to adapt legacy C++ code to our dialect. Since porting consumes precious developer time, we measure the impact of such efforts in §2.5. Our evaluation shows that the trade-off between adopting type++ and the new security guarantees is acceptable.

Phantom casting. Listing 2.2 shows an example of phantom casting. A phantom class is a parent-child relationship where the child's data layout is equivalent to the parent's data layout [59] (lines 3 and 5). Even though this corresponds to an illegal downcast in principle, in practice, current C++ implementations ignore phantom casting to maintain interoperability between C and C++ code.

Solution and Impact: Even though this practice may be exploited for de-facto type confusion [77], type++ allows their usage for backward compatibility. However, since this may cause issues in later development, we implement a static analysis to find active cases. In our evaluation, we report no such type confusion when applying Property 2 (§2.5.1).

`sizeof()/offsetof()` usage. C++ offers the `sizeof()` and `offsetof()` operators. The former checks the size of classes or structures. The latter, instead, returns the offset of a field with respect to a class or structure. In Listing 2.3, we show an example as part of a `Substitution Failure Is Not An Error` (SFINAE) expression [58]. Adopting type++ introduces an extra

```
1 #include <stdio.h>
2
3 class BaseType { /* other fields */ };
4
5 class PhantomType : BaseType {};
6
7 void checkCast() {
8   BaseType *ptr = new BaseType();
9   // The following cast results in Undefined
10  // Behavior in standard C++ but is commonly
11  // tolerated as both classes have the same layout.
12  if (dynamic_cast<PhantomType*>(ptr) == NULL)
13    printf("error!\n");
14 }
```

Listing 2.2 – Example of *phantom* casting

RTTI pointer in the classes `yes` and `no` (lines 8 and 12), altering their expected size and the results of `val` (line 28).

Solution and Impact: `sizeof` and `offsetof` return the correct values when taking the extra type field into consideration, which is the expected behavior for most use cases (*e.g.*, `malloc(sizeof(T))`). However, issues may arise when comparing the results with a scalar or with another value returned by these operators. Comparison with a scalar (*e.g.*, `sizeof(no)` == 2) is already discouraged in standard C++ as the type size might be implementation-dependent. With type++'s extra type field, the expected scalar will differ. To mitigate the misuse of these features, we design two strategies. First, one can enforce Property 2 to reduce the number of instrumented classes, thus improving the cross-compatibility between standard C++ and type++. Second, we designed a static analysis to emit a warning whenever an instrumented class is used as a parameter of `sizeof()` or `offsetof()`. Our evaluation shows high usage of this pattern, specifically 957 occurrences for Property 1 and 129 occurrences for the relaxed Property 2 in SPEC CPU benchmarks. In practical terms, a single pattern (*e.g.*, a SFINAE template) required a trivial source code adaptation. This high compatibility is due to the small number of scenarios where `sizeof()` misbehave (*i.e.*, when comparing to a scalar or when padding is involved as in Listing 2.3). The code adaptations are also usually simple (*e.g.*, increasing the size of `no` to over 16 bytes, line 12). We discuss these cases in more detail in §2.5.1.

Implicit `placement new`. Listing 2.4 shows an example of *implicit* `placement new`. The class Y is used to allocate memory for X in a similar fashion to the `placement new` C++ operator. Without type++, the objects instantiated by classes X and Y have the same size (line 11) and may be used interchangeably—as long as the developer accepts the underlying type confusion. However, in type++, due to the Property 1, class X and class Y contain an RTTI pointer which increases the class size. Therefore, the resulting size of class Y exceeds the one of class X as it includes space for two vtable pointers, its own mandated by RTTI and the one contained in

```
1 #include <iostream>
2
3 struct foo {
4   typedef float X;
5 };
6
7 struct yes {  // type++: sizeof(yes) == 16
8   char c[1]; // due to struct padding matching
9 };           // the alignment of the \acrshort{RTTI} pointer.
10
11 struct no {  // type++: sizeof(no) == 16
12   char c[2]; // due to struct padding matching
13 };           // the alignment of the \acrshort{RTTI} pointer.
14
15 template <typename T>
16 struct has_typedef_X {
17
18   template <typename C>
19   static yes& test(typename C::X*);
20   template <typename>
21   static no& test(...);
22
23   static const bool val =     sizeof(test<T>(nullptr)) == sizeof(yes);
24 };
25
26 int main(int argc, char *argv[]) {
27   std::cout << std::boolalpha;
28   std::cout << has_typedef_X<int>::val << std::endl; // standard C++: false, type++: true
29   std::cout << has_typedef_X<foo>::val << std::endl; // true for both standard C++ and type++
   ↪
30   return 0;
31 }
```

Listing 2.3 – Code example of `sizeof()` in SFINAE. The issue arises because `sizeof(yes)` and `sizeof(no)` now have the same size due to extra padding when using type++.

`__blob_` (line 4) for the vtable pointer of X. type++ reports the type confusion between class X and class Y but might produce an error at runtime as the cast at line 9 leads to out-of-bound accesses.

Solution and Impact: As classes X and Y are not related, type++ alerts the developer of the type confusion. The valid fix would be to replace this pattern with the C++ `placement new` operator. In type++, this pattern cannot be handled automatically and requires a fix for valid execution. While current C++ compilers accept this pattern, it relies on Undefined Behavior as the translation between X and Y is not well-defined. In our evaluation, we found 131 (for Property 1) and 3 (for Property 2) instances of this pattern in the 2,040 kLoC of the SPEC CPU benchmarks. However, none of them resulted in a runtime error nor produced unexpected behaviors (*i.e.*, the benchmarks' output was correct). In addition to reporting the type confusion, type++'s analysis identifies this code pattern and warns the developer that a fix is necessary.

```
1 class X { /* other fields */ };
2
3 class Y {
4   char __blob_[sizeof(X)];
5 };
6
7 int main(int argc, char *argv[]) {
8   X* x;
9   Y* y = reinterpret_cast<Y*>(x);
10  X* z = reinterpret_cast<X*>(y);
11  static_assert(sizeof(X) == sizeof(Y), "error");
12  // The above assert is true in standard C++ but
13  // false in type++.
14  return 0;
15 }
```

Listing 2.4 – Code example of implicit `placement new`

2.4 type++ technical details

2.4.1 Default constructors

To enforce the above properties, type++ requires a default constructor for each instrumented class to set the vtable pointer of an object correctly. We use the default constructors for handling heap allocators (§2.4.3) and union (§2.4.4) initialization. However, naively synthesizing default constructors might break either the C++ semantic or the original program logic. For instance, a class might be purposely designed without a default constructor to be a Plain Old Data (POD) type or a non-clonable object (*i.e.,* classes without copy constructor [113]).

To automatically inject dummy default constructors without breaking either the C++ standard nor the developer intention, type++ uses a two-step compilation approach. The first step verifies that the program is C/C++ compliant without considering the type++ specifications, *i.e.,* ensuring that the program adheres to the C/C++ semantic and is free of compilation errors. In the second step, we override the C++ semantics and forcibly inject the default constructors. Note that, if the first step fails, the compilation does not proceed. This allows us to keep the original C/C++ language semantic along with helpful compiler warnings or error messages while synthesizing default constructors for the vtable pointer initialization.

2.4.2 Uninitialized variables

Uninitialized variables lead to undefined behavior [127]. An uninitialized object does not call a constructor and therefore does not initialize the vtable pointer. This might cause crashes during

`dynamic_cast` as well as crashes during type++ checks. We detect the use of uninitialized variables by enabling the Clang flag `-Wuninitialized` during the initial compilation and report the warnings to the developer.

2.4.3 Allocation through C-style allocators

When an object is created using an allocator from the C-style `malloc` family (*e.g.,* `malloc`, `realloc`, or `calloc`), the system only reserves space for the object without initializing it, thus not setting the vtable pointer. To enforce Property 1, our static analysis automatically identifies the use of C-style allocators for instrumented classes. Then, it explicitly sets the vtable pointer right after the allocation using a default constructor. Spurious use of `calloc` instead of `malloc` may allocate more memory than required for a single object. We mitigate this behavior by retrieving the size of the block of memory effectively allocated via `malloc_usable_size()` and loop through the allocated memory, calling the constructor for as many objects that fit in the block. This transformation allows us to correctly handle vtable pointer initialization through `realloc` by only setting the vtable pointer on newly allocated memory. Indeed, calling the constructor on previously allocated memory could result in data being overwritten. Therefore, when faced with an allocator from the `realloc` family, type++ first retrieves the size of the previously allocated memory. This allows us to compute the range of newly allocated memory and only initialize the vtable pointer there. We process `std::allocator_traits` [113] likewise: For each class implementing the trait, we identify the memory allocation and inject the vtable pointer initialization, accordingly. While our prototype implementation relies on `libc` to retrieve the actual allocated size, extending support to other systems is straightforward (e.g., via `_msize` on Windows [129] or `malloc_size` on Mac OS X [3]).

Besides the standard system allocators, we encountered many custom allocators built on top of the `malloc` primitives. Many of them injected a custom header in front of the allocated object to store metadata or debug information. Having an extra header obscures the object's location, thus hindering the pointer arithmetic and causing unpredictable crashes. To automatically handle these cases, type++ contains an allow-list with the custom allocator functions and their respective header size. The compiler takes this number into account to locate the future objects in memory and then invokes the constructor accordingly. The allow-list is externally configurable. We use this technique to correctly handle POV-Ray (SPEC CPU2006 and CPU2017), Xalan-C++, and Blender, which otherwise would generate non-protected objects (*i.e.,* missing vtable pointer during type checks). While our technique handles the benchmarks correctly, we recommend a rewrite of these allocator patterns, following a modern programming style [86].

2.4.4 Polymorphic union members

In type++, a union, like any object, must have a type attached according to Property 1. By design, it is unknown which type a union will hold at compilation time. type++ cannot, therefore, know which constructor to call when the union is first declared. This issue also prevented C++, before the C++11 standard, from having polymorphic objects as union members. To highlight the issue, let us suppose we have a union with two polymorphic classes and an `int` variable. When we create an object, we cannot predict which member's constructor to call as we do not know which member of the union will be activated later. Explicit constructors (*e.g.,* `placement_new`) are generally needed when the type is changed through a union [24]. To automatically address this, our compiler inserts a constructor call every time a union switches from or to an instrumented type.

2.4.5 Initialization of `const` variable

`const` variables have to be completely initialized during their declaration [114]. In type++, the object initialization is done in two steps: first, we set the vtable pointer, and second, we invoke the actual object constructor. Therefore, naively applying this transformation would break the `const` constraint. In our compiler, we solve this case by relaxing the `const` property for classes instrumented and allowing exactly two initialization steps. First, forcibly inserting an additional constructor exclusively sets the vtable pointer. Then relying on the native constructor to instantiate the object. The language semantics are preserved as program correctness is verified in the first compilation step described in §2.4.1. As an alternative, we could predict the initialization of the `const` variable and initialize it in a single step, however, this requires a complex error-prone analysis.

2.4.6 Interaction with other C++ libraries

Programs sharing classes with external libraries must agree on a common per-object memory layout. For instance, a program P might use a class C to communicate with a library L. Compiling P for type++ adds an extra vtable pointer to C and modifies its memory layout (due to Property 1). Therefore, if L is not aware of the new layout of C, the program will execute incorrectly.

To solve this issue, we simultaneously build P and L with our type++ compiler and impose Property 1 to both. This approach ensures P and L follow the same data layout. For Property 2, we instrument the set of classes involved in casts in either P or L. However, this approach requires each library L to be specialized for each program P. In case L is closed-source, one can tune Property 3 and select only those classes that do not interact between P and L. This reduces the security guarantees as all the objects of the excluded classes will not be checked at runtime

but improves the compatibility with closed-source libraries. When possible, we highly recommend specializing the library to avoid any risk of type confusion.

2.4.7 Interaction with the kernel and non-C++ code

Similar to uninstrumented C++ libraries, interacting with the operating system and other native code via modified objects may create compatibility issues. Considering the burden of rebuilding the kernel, we propose a wrapper function for system calls in `libc` to remove the vtable pointer before sending modified objects to the kernel. Once the system call returns, the wrapper then adds back the vtable pointer for further handling by the compiled program.

Due to Property 2, we did not encounter any such case in our evaluation. All type++ programs interacted with `libc` without requiring any wrapper. While this is not a guarantee, the large amount of benchmarks we executed indicates that this is not a problem in practice. Without Property 2, each C struct, including those passed to the kernel, would be instrumented, requiring rebuilding the kernel or inserting multiple wrappers to adjust the objects' layout and remove vtable pointers. This highlights the benefits of our optimization over the coarse-grained approach of Property 1.

In general, if such translated structs are passed to non-type++ code, a compiler pass can detect it and warn the programmer that a translation function may be required.

2.4.8 Prototype implementation

Our type++ compiler prototype is based on the LLVM infrastructure (version 13.0.0, the latest release at project instantiation). The implementation consists of 3.1 kLoC added to Clang and LLVM passes. In particular, we modify Clang/LLVM to (i) detect and warn in case of incompatible code patterns, and (ii) change the data layout in non-polymorphic objects. We implement type-casting verification by extending the security properties of LLVM-CFI [120]. Natively, LLVM-CFI performs type checks only over polymorphic objects, thus leaving a wide attack surface for all non-polymorphic object types. In type++, we augmented all classes with RTTI, thus extending the coverage of LLVM-CFI.

As part of the evaluation, we create a microbenchmark to compare the overhead of different typecasting verification approaches. As described in §2.5.3, the workload tries to mimic the typecasting behavior of OMNeT++ which is part of the SPEC CPU2006 benchmark.

2.5 Evaluation

Our evaluation of type++ explores four research questions:

(**RQ1**) What is the compatibility between C++ programs and the type++ dialect (§2.5.1)?

(**RQ2**) What are the new type++ security guarantees (§2.5.2)?

(**RQ3**) What is the overhead introduced by type++ (§2.5.3)?

(**RQ4**) What is the porting effort of type++ for a major C++ project (§2.5.4)?

Experimental setup. Our experiments are performed in a Docker container based on Ubuntu 20.04 running on a server with two Intel Xeon E5-2680 v4 @ 2.4GHz CPUs with 256GB of RAM in total.

Target programs. We evaluate type++ on two popular benchmarks: SPEC CPU2006 [50] and CPU2017 [13]. For each benchmark, we select all included C++ programs, resulting in seven targets from SPEC CPU2006 and nine from SPEC CPU2017. We additionally target Chromium 90, the most recent version compatible with LLVM 13 that is available on Debian. We measure Chromium runtime performance through JetStream2 [8], an aggregation of JavaScript and WebAssembly benchmark.

State-of-the-art. We compare type++ against four recent state-of-the-art projects: TypeSan [49], HexType [59], EffectiveSan [30], and LLVM-CFI [120]. Note that LLVM-CFI provides typecasting verification but limits it to polymorphic objects only. Since TypeSan, HexType, and EffectiveSan are seven and six years old, respectively, we managed to compile them only against the seven targets from SPEC CPU2006, while we built all the 16 targets with LLVM-CFI. We run all the related work experiments with the experimental setting of type++. As a common baseline, we use LLVM 13 and optimization level `O2` with Link Time Optimization enabled as it is required by LLVM-CFI. LLVM-CFI and type++ are both built on top of LLVM 13. The relative numbers of both of them refer to native execution on LLVM 13. Likewise, we used LLVM 3.9 for TypeSan and HexType and LLVM 4.0 for EffectiveSan since they were developed for these platforms, respectively.

2.5.1 Compatibility analysis

We perform a compatibility analysis and count lines of code (LoC) that match the programming patterns described in §2.3.1. For detecting *implicit* `placement new` and `sizeof()/offsetof()`, we employ a conservative static analysis over the AST to avoid true negatives, as done in previous works [27]. For phantom casts, we extract the class hierarchy and infer the classes' data layout through LLVM. Since we rely on static analysis, the final numbers are a generous overapproximation. We implement these analyses as a plugin for Clang. Additionally, we count the

Table 2.1 – Number of warnings in SPEC CPU2006 and CPU2017. For each cell, we indicate the number of possibly incompatible LoC in the format (# LoC w/ Property 1) / (# LoC w/ Property 2). The numbers show that most of the programs do not require modification, while the smallest patch modifies only 2 LoC (deal.II) and the biggest only 131 LoC (POV-Ray). This evaluation confirms the low impact of porting C++ projects to type++ dialect. The last three columns indicate if the program uses a Custom Memory Allocator (CMA) (*i.e.,* C), the number of unique program locations that introduce type confusions, and the compilation duration overhead, respectively.

	Use Case	LoC	Implicit `plac. new`	Phantom Casting	`sizeof()` `offsetof()`	Uninit. Objects	LoC Changed add	LoC Changed del	CMA	T. C. Errors	Comp. Overhead
SPEC CPU2006	NAMD	3K	0 / 0	0 / 0	1 / 0	0 / 0	-	-	-	-	106%
	deal.II	94K	1 / 0	3 / 0	51 / 0	0 / 0	2	-	C	-	94%
	SoPlex	28K	15 / 12	0 / 0	3 / 0	0 / 0	-	-	-	-	103%
	POV-Ray	78K	0 / 0	0 / 0	223 / 61	9 / 3	79	44	C	56	106%
	OMNeT++	26K	0 / 0	14 / 1	7 / 1	0 / 0	-	-	C	-	93%
	Astar	4K	0 / 0	0 / 0	9 / 0	0 / 0	-	-	-	-	112%
	Xalan-C++	264K	5 / 0	129 / 0	13 / 0	1 / 0	-	-	C	4	107%
SPEC CPU2017	CactuBSSN	63K	0 / 0	0 / 0	1 / 0	0 / 0	-	-	-	-	108%
	NAMD	6K	0 / 0	0 / 0	6 / 0	0 / 0	-	-	-	-	110%
	Parest	359K	72 / 20	7 / 0	101 / 3	0 / 0	-	-	-	1	108%
	POV-Ray	80K	0 / 0	0 / 0	223 / 61	9 / 3	87	44	C	53	109%
	Blender	615K	14 / 0	0 / 0	15 / 9	0 / 0	47	11	C	1	103%
	OMNeT++	85K	0 / 0	102 / 0	35 / 1	2 / 0	-	-	C	2	111%
	Xalan-C++	291K	24 / 3	125 / 0	259 / 1	0 / 0	-	-	C	5	109%
	Deep Sjeng	7K	0 / 0	0 / 0	7 / 0	0 / 0	-	-	-	-	117%
	Leela	30K	0 / 0	0 / 0	3 / 0	0 / 0	-	-	-	-	84%
	Total	2,040K	131 / 35	380 / 1	957 / 137	21 / 6	215	99	-	122	106%

number of uninitialized objects that are undefined behavior in standard C++. These issues are orthogonal to the type++ porting efforts and may result in miscompilation even in standard C++. To identify uninitialized objects, we enable the flag `-Wuninitialized` during the vanilla compilation [21]. We measure the number of affected code patterns for Property 1 and Property 2, respectively.

Table 2.1 shows the result of our analysis. Overall, compiling the SPEC benchmarks with Property 2 produces 179 warnings compared to 1480 warnings for Property 1, *i.e.,* the optimization of only instrumenting cast-related classes reduce the number of warnings by almost 90%. When applying Property 1, we encounter a few incompatibilities with the `sizeof()` operator as part of SFINAE expressions in the `Boost` library (similar to the example in Listing 2.3). As the classes involved are never cast, the issues disappear when using Property 2. Generally, most of the programs work without any modifications and only a fraction of them require manual source code adaptations. Well-behaved programs written in modern C++ generally do not require any modifications. More specifically, we modify deal.II, Blender, and POV-Ray (in both SPEC CPU2006 and CPU2017). They require changing 314 LoC in total (around 0.04%). Besides deal.II, whose modification was trivial, we discuss Blender porting efforts below and the ones of POV-Ray and Xalan-C++ in Appendices §A.2 and §A.3.

Focusing on the detected warnings, *implicit* `placement new` warnings require to inspect at maximum 131 LoC when considering Property 1 and only 3 LoC in the case of Property 2. For *phantom casting*, our analysis highlights 129 LoC for Property 1 but none when Property 2 is used. `sizeof()/offsetof()` warnings concern at most 957 LoC with Property 1 and only 129 LoC otherwise. In this case, the only real incompatibility observed with `sizeof()` resides in SFINAE expressions in the `Boost` library. The other warnings for `sizeof()` are tied to memory allocation routines, thus not causing any trouble as the size is correctly adjusted during compilation. We do not observe any runtime errors/misbehavior related to warnings from `offsetof()`. We also encounter a very limited number of uninitialized objects, 21 for Property 1 and 6 for Property 2, respectively. Upon further investigation, we conclude that the warnings were false positives due to the over-approximation of Clang's default analysis and do not introduce runtime issues, such as object initialization behind opaque conditions.

Finally, regarding custom allocators, we address them by using an allow-list of functions. During our evaluation, we find 16 custom allocators across 6 programs (deal.II, Blender, SoPlex, Xalan-C++, and both versions of POV-Ray). type++ gracefully handles allocators with and without custom headers within the same program, for example, `MEM_mallocN` and `BLI_memarena_alloc` in Blender.

Use case: Blender. We encounter three different sources of incompatibility when porting Blender. The first problem involves shared structures (defined in `.h` files) between C and C++ code (§2.4.7). Due to Property 1, the `structs` in C++ now contain an additional `vptr` field, that remains absent in the C code. This results in having two different memory layouts for the `structs` in C and type++ code, leading to unpredictable crashes. We fix this issue by adding a field of the same size as the `vptr` pointer at the beginning of the structure in the C code, the field is activated only when compiling as C source code, thus adjusting the memory layout between the two languages. Our second issue is that Blender relies on fat pointers to store metadata information (*e.g.,* using the LSB as object type reference). The Blender developers employ bit-mask operations before pointer dereferencing. type++ flags these locations as type confusion as it cannot locate the vtable pointer due to the address offset. Upon closer inspection, we observe the vtable pointer is correctly set but the memory address is not pointing to the beginning of the object. We manually patch the code by adding a bit-mask before cast operations to adjust the pointer operations. If fat pointer casting is used at multiple locations, applying the unmasking operation automatically at cast verification time could be implemented as an optional feature. As overloading pointers is not common, we argue in favor of manually fixing a few cases. Blender casts fat pointers only at two locations, we address this issue with 20 LoC modified. Finally, Blender uses custom heap allocators that wrap the standard `malloc` family functions. In this case, we add the `malloc`-like functions in type++'s custom allocator list without source code modification (§2.4.3). Overall, we modify only 58 LoC out of more than 600K (less than 0.1%) while the program logic stays untouched.

Table 2.2 – Overhead and coverage evaluation of type++ and LLVM-CFI deployed over SPEC CPU2006 and CPU2017. For each program, we indicate the average overhead measured (*i.e.,* %), the number of down-to casts, and unrelated casts observed in our experiments. In the right-most part of the table, we compare LLVM-CFI and type++: the delta columns (Δ) show the difference in terms of casts protected, while the last two columns indicate type++ memory overhead.

		LLVM-CFI			type++			Δ		memory (%)	
	Use Case	(%)	down-to	unrelated	(%)	down-to	unrelated	down-to	unrelated	avg.	max.
SPEC CPU2006	NAMD	-0.52	0	0	-0.82	0	0	0	0	0.58	0.59
	deal.II	1.50	0	0	2.02	17,462M	122M	17,462M	122M	-0.18	0.20
	SoPlex	-0.22	0	0	1.15	209K	28M	209K	28M	1.94	2.98
	POV-Ray	0.55	0	1K	4.11	11,477M	1,342M	11,477M	1,342M	0.44	0.43
	OMNeT++	3.43	1,897M	3	4.07	2,521M	270K	624M	270K	0.35	0.16
	Astar	-1.16	0	0	-0.15	0	0	0	0	0.27	-0.09
	Xalan-C++	-0.12	282M	4K	0.09	284M	5K	2M	612	0.32	-0.01
SPEC CPU2017	CactuBSSN	-2.54	30	0	-0.87	80K	100	80K	100	0.12	0.11
	NAMD	-0.47	0	0	-0.42	0	0	0	0	0.17	0.17
	Parest	-0.26	11M	18K	-0.11	2,462M	85M	2,451M	85M	-0.83	-0.58
	POV-Ray	0.66	0	573	3.04	45,861M	5,370M	45,861M	5,370M	3.82	3.82
	Blender	0.25	0	0	4.58	0	7M	0	7M	1.39	1.37
	OMNeT++	3.22	1,428M	6M	2.59	2,786M	6M	1,358M	829K	1.21	1.10
	Xalan-C++	-0.28	227M	4K	-0.81	227M	198M	0	198M	1.52	1.20
	Deep Sjeng	0.12	0	0	-0.41	0	0	0	0	0.00	0.00
	Leela	0.57	0	0	0.43	21M	1K	21M	1K	0.12	0.16

Takeaway. This evaluation shows the high compatibility between C++ and type++. In particular, applying Property 2 requires limited effort to port C++ projects, *i.e.,* we modified 123 and 131 LoC for both POV-Ray (0.16%) and 58 LoC for Blender (< 0.1%) while it protects every cast operation. Refer to §2.6 for specifics that could limit type++ adoption.

2.5.2 Security evaluation

To evaluate the security guarantees of type++, we measure the number of downcasts checked at runtime. Specifically, we run the 16 use cases of the SPEC benchmarks against type++, TypeSan, HexType, and LLVM-CFI whose results are in Table 2.2 (LLVM-CFI) and Table 2.3 (TypeSan and HexType).

Table 2.2 shows the result of our experiments in regards to coverage of type casts with type++ compared to LLVM-CFI. Due to the introduction of Property 1, type++ allows all previously undetected objects to emerge and be properly verified. This is particularly evident for SoPlex, POV-Ray, and Leela, in which type++ alters normal classes into polymorphic ones. type++ can check the integrity of all downcasts, resulting in an additional 13B and 51B runtime cast for POV-Ray 2006 and 2017, respectively. Similarly for Leela, where type++ can now monitor the integrity of 21M down-to-casts that otherwise would remain unchecked.

Table 2.3 – Performance comparison of type++ against TypeSan, HexType, and EffectiveSan. Since TypeSan and HexType only mitigate down-to-casts, we do not report unrelated casts for type++. EffectiveSan has a different definition of cast checking which does not allow a direct comparison. We, therefore, omit the numbers and include a discussion in §2.5.3.

	Use Case	TypeSan (%)	TypeSan # cast	HexType (%)	HexType # cast	Eff.San (%)	type++ (%)	type++ # cast
SPEC CPU2006	NAMD	0.00	0	0.17	0	588	-0.82	0
	deal.II	74.25	3,379M	6.13	3,380M	1,212	2.02	17,462M
	SoPlex	0.00	0	0.00	209K	497	1.15	209K
	POV-Ray	23.52	0	-2.39	0	667	4.11	11,477M
	OMNeT++	44.32	2,014M	29.21	2,014M	229	4.07	2,521M
	Astar	1.70	0	1.05	0	310	-0.15	0
	Xalan-C++	35.46	283M	17.96	283M	1,593	0.09	284M

Table 2.4 – Type confusions found by HexType, EffectiveSan, and type++ in SPEC CPU. In SPEC CPU2006, type++ finds a superset of the errors found by previous works while incurring lower overhead.

	Use Case	HexType	Eff.San	type++
2006	POV-Ray	0	56*	56
	Xalan-C++	2	2*	4
2017	Parest	-	-	1
	POV-Ray	-	-	53
	Blender	-	-	1
	OMNeT++	-	-	2
	Xalan-C++	-	-	5

* Numbers from the paper that we could not reproduce.

Using inline cast information, combined with a complete list of custom allocators (§2.4.3), allows type++ to overcome the coverage issues affecting disjoint metadata approaches [59]. As shown in Table 2.3, type++ does not miss any cast and indeed protects *every* object passing through a downcast. Similarly to LLVM-CFI, type++ has an option to additionally protect unrelated casts (*i.e.,* cast over `void*`) whose classes are polymorphic or instrumented. This feature allows type++ to stretch its protection beyond the disjointed approaches without any further porting cost. As a consequence, type++ protects a vast number of runtime casts that, so far, were unprotected, *e.g.,* for SoPlex and POV-Ray 2017 (28M and 5B casts respectively).

There are limited actions to bypass type++, *e.g.,* a possible issue could emerge if custom allocators are not properly allow-listed by the developer. Another possibility is to use a flawed type check logic. We assume the logic is sound (§2.2), moreover, our prototype relies on the standard type checks in LLVM, which has been widely tested and optimized by the community. Finally, another cause of type confusion could come from undefined behavior or other memory safety violations, which we considered out of scope (§2.2).

From our evaluation, we observe a total of 122 type confusions in SPEC CPU2006 and CPU2017, among them, 14 bugs are newly discovered by type++. Table 2.4 compares the type confusions found by type++, HexType, and EffectiveSan. type++ can mitigate all the bugs discovered by either EffectiveSan or HexHype, demonstrating that our dialect protects an attack surface covering both state-of-the-art tools. We discuss how we tackle type confusions in POV-Ray (Appendix §A.2) and Xalan-C++ (Appendix §A.3). We attach detailed technical reports describing the type confusions in our open-source documentation. These type confusions have been fixed in more recent versions of the benchmark code. The patches are in line with type++'s properties (*e.g.,* using a proper class hierarchy and dynamic cast).

Takeaway. Our evaluation shows that type++ validates *every* downcast: it covers up to 10^9 more casts compared to the previous state-of-the-art, *e.g.,* deal.II and POV-Ray have 17B and 45B more runtime checks than with LLVM-CFI.

2.5.3 Performance overhead

We assess the performance overhead of type++ over the two benchmark sets previously introduced, SPEC CPU2006 and CPU2017. For the two SPEC benchmarks, we recompiled each use case in vanilla (*i.e.,* using the unmodified Clang as the compiler) and with alternative protection/mitigation techniques, specifically, TypeSan, HexType, EffectiveSan, and LLVM-CFI. We repeated each run five times and considered the average execution time (we observed a negligible standard deviation). For type++, we rely on Property 2 as it offers the same security guarantee as Property 1 while instrumenting fewer objects, reducing performance overhead. We do not evaluate Property 1 overhead as it would require analyzing a vastly superior number of warnings as highlighted in §2.5.1 to obtain a worse runtime performance and no additional security guarantees. We summarize the evaluation of type++ against LLVM-CFI in Table 2.2, while the results against TypeSan and HexType are in Table 2.3. In both tables, we show the runtime overhead against their baseline (*i.e.,* %) and count the number of casts protected at runtime. The latter is further split between the down-to cast in the scope of type++ and the unrelated cast that LLVM-CFI and type++ additionally support. We consider only down-to-cast for TypeSan and HexType since they do not cover unrelated casts. For EffectiveSan, we omit the number of casts since their definition is incompatible with ours. Additionally, we compare the memory overhead of type++ against LLVM-CFI in the last two columns of Table 2.2. We also investigated the impact of patches on the program's performance. For this, we compile both original and patched programs against vanilla Clang and measure the overhead. Finally, we break down the cost of each operation of the different type-checking approaches. This shows that disjoint metadata approaches suffer from heavy lookup costs and cannot achieve performance on par with inlined approaches.

The overhead introduced by type++ (Table 2.2) ranges from -0.87% (CactuBSSN) to 4.58% (Blender), while the LLVM-CFI overhead stays between -1.16% (Astar) and 3.22% (OMNeT++

2017). The type++ overhead loosely correlates with the number of additional casts protected. For instance, OMNeT++ 2006 shows a performance overhead of 4.07% while protecting around 800M more casts than LLVM-CFI when considering *down-to-cast* and *unrelated casts* together (around 2.5B casts in total for type++ against 1.9B casts for LLVM-CFI, *i.e.,* 30% more). Similarly, for POV-Ray 2006, LLVM-CFI introduces an overhead of 0.55% for protecting only 1K casts, while type++ introduces a 4.11% overhead by covering more than 12B casts in total (down-to and unrelated casts together). This is due to our core property that ensures that every object possesses type information queryable at runtime, allowing type++ to stretch inline protections already tested and optimized to every cast, including many that remained unchecked by previous works.

In comparison with disjoint metadata structure approaches, such as HexType and TypeSan, the benefits of type++ are even more noticeable (Table 2.3). While HexType exhibits a maximum overhead ranging up to 29.21% (OMNeT++) and TypeSan 74% (deal.II), type++'s overhead is limited to 4.58% and 4.11% for Blender and POV-Ray 2006, respectively. Nonetheless, type++ is capable of protecting more casts than previous approaches, *i.e.,* type++ covers 25% more casts than HexType for OMNeT++. Furthermore, Property 1 allows us to protect every cast, resulting in 14B casts from deal.II 2006 protected, that are covered by neither HexType nor TypeSan. The performance improvement is mainly driven by type++ not requiring heavy cast tracking nor disjoint metadata structures operations that introduce a notable overhead.

EffectiveSan uses Low-fat pointers [67] to store type information. This work shares a different threat model compared to type++ since they trace any type of cast regardless of security implications. The result is an impactful overhead that ranges from ~230% to ~1600%. This exemplifies how Property 2 limits the overhead by focusing on real cast operations. Additionally, we observe a few false positives when deploying EffectiveSan over deal.II. Specifically, EffectiveSan wrongly reports as type confusion some template variables that it infers belong to different types. Conversely, type++ did not show any false positives in our evaluation.

In terms of memory, type++ introduces a negligible overhead compared to LLVM-CFI that stays below 1.20% on average and 1.52% at maximum. The only exception is POV-Ray 2017 with an overhead of 3.82% (average and maximum). We deem this (limited) discrepancy to be caused by the additional casts protected compared to LLVM-CFI. Since we protect more objects, we introduce more RTTI in memory. However, we consider the observed overhead acceptable in practice.

The impact introduced by our patches is less than 3% in the worst case—1.46% (deal.II), 1.43% (POV-Ray 2006), 2.83% (POV-Ray 2017), 2.11% (Blender). Therefore, we argue that the patches, while modernizing the program code, do not harm performance nor reduce type++ overhead.

Table 2.5 – Breakdown of the cost [ns] of each operation for disjoint and inlined metadata type checking. For HexType, verification (Verif.) is the sum of a Type check and a Lookup operation. A cast verification is 7.7× faster in type++.

HexType					LLVM-CFI/type++
Insert	Delete	Lookup	Type check	Verif.	Verification
19.17	3.21	2.25	2.88	5.13	0.66

Finally, Astar, Deep Sjeng, and NAMD are examples of programs without any casting operations. In these cases, type++ did not meaningfully affect their performance since we do not introduce any cast-checking nor modify any object.

Performance on a microbenchmark. To more precisely understand the causes of the performance overhead for each type confusion protection, we design a microbenchmark that separates the cost of each operation. In particular, we analyze HexType as an example of the disjoint metadata approach. HexType can be decomposed into four major operations: metadata insertion, metadata deletion, metadata lookup, and type checking. We compare these operations with the ones of both LLVM-CFI and type++ as they are identical. As the type information is directly stored in the object the lookup cost is minimal, we, therefore, concentrate our effort on evaluating the cost of the LLVM-CFI type check. Despite investing large efforts, we were unable to isolate the different costs of the EffectiveSan prototype implementation. The prototype injects extra optimization flags into the compilation pipeline which activates vector optimizations as part of its LLVM integration. These extra optimizations disturb the results compared to the baseline. We were unable to disable these extra optimizations.

Our microbenchmark mimics the OMNeT++ workload from SPEC CPU2006: it creates 480M objects, 45% of which are involved in cast operations, and executes 2.5B cast operations in total. Per our analysis, 99% of the casts in OMNeT++ are caused by five unique code locations that we replicate in our microbenchmark. We also approximate the cast distribution and the class hierarchies. The most complex cast has a five-level hierarchy between the instantiated object and its base class. We measure the execution order of the 900M cast operations by looping through the five cast operations. The code is written to minimize caching effects and assess the actual cost of each operation. The benchmark is compiled with O2 as optimization level and evaluated on the same machine as the rest of type++ evaluation.

For LLVM-CFI, we measure the duration of the validation of a static cast that type++/LLVM-CFI instrument. For HexType, instead, we isolate and measure the four main operations of disjointed metadata approaches: insert, lookup, delete, and type check. Table 2.5 summarizes the results. HexType has additional costs to handle the metadata lifetime (Insert and Delete), which is a single write for type++. Cast verification is a lookup and a type check for disjoint metadata approaches, while it is only a RTTI verification for type++. The figures highlight the heavy cost

of the lookup operation of disjoint metadata approaches, which is more than three times slower than the type check itself of inlined metadata. This result shows that the key limitation of disjoint metadata approaches cannot be resolved by faster type checks alone. In disjointed metadata approaches, the bottleneck is caused by the query time to retrieve the data structures containing the type information necessary for the check. It has already been optimized across the different previous works. Currently, type checks are seven times slower in HexType than in type++. In addition, current disjoint data structure implementations are not thread-safe, enforcing thread-safety would likely introduce higher overhead. This leads us to conclude that type++ is the only reasonable approach to mitigate type confusion bugs while maintaining reasonable performances, as also shown by our experiments in Chromium (§2.5.4). Further improvements on the performance of `dynamic_cast` are possible as shown in [39, 83], increasing, even more, the performance advantage of type++ over disjoint metadata approaches.

Takeaway. type++ largely outperforms state-of-the-art approaches in terms of overhead, while it extends inline-optimized protections to cast locations that would not be covered otherwise.

2.5.4 Use case: Chromium

In this section, we showcase the porting of Chromium to type++. We choose this project as an example of an established legacy codebase, analyze the challenges, and compare our findings with state-of-the-art solutions. Chromium is the open-source project underlying Google Chrome, the most popular browser. Chromium frequently faces type confusion vulnerabilities (*e.g.,* CVE-2019-5757, CVE-2020-6464, CVE-2022-3315). With over 35 MLoC written in C++, it is one of the biggest active open-source C++ projects and, therefore, an ideal target for type++. We choose Chromium version 90, which is distributed with Debian 10. For performance reasons, Chromium developers never use `dynamic_cast`, resorting to the unsafe `static_cast` in release builds. However, this performance trade-off comes at a security cost. Applying type++ to Chromium improves the browser security and serves as a benchmark for type++'s real-world applicability.

In the rest of this section, we detail the result of our compatibility analysis and discuss our deployment strategy. Then, we describe the patches applied, measure the overhead, and compare our approach to other mitigations.

Compatibility analysis. To assess the compatibility of Chromium with type++, we execute our analysis with Property 2, which reports 3,339 warnings (§2.5.1). 54.1% of these warnings were linked to the implicit `placement new` issue, but none required a code change. The warnings, however, correctly point to code segments where improvements were sensible. While our analysis did not report any phantom class, it highlights 1,530 locations where `sizeof` was used.

Another source of potential incompatibilities comes from Protobuf [43], Google's data format for serialized structured data. `protoc`, the Protobuf compiler, generates C++ code from

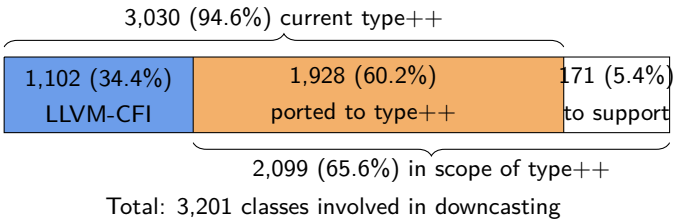

Figure 2.1 – Chromium contains 3,201 classes involved in downcasting, among which 2,099 are in necessary for type++ to achieve Property 2. LLVM-CFI already protects 1,102 classes (34.4%). type++ currently supports 1,928 additional classes in Property 3 (60.2%, 91.8% in respect to "in scope for type++"). 171 classes (5.4%) are yet to be ported. Figure not to scale.

specifications. type++ analysis reported code patterns in generated code which required a minor adaptation to `protoc` to ensure that no infringing code is generated.

type++ deployment strategy. Figure 2.1 illustrates the Chromium classes involved in downcasting operations and which fractions type++ and LLVM-CFI each protect. Enforcing Property 2 over Chromium requires instrumenting 2,099 classes and manually addressing 3,339 warnings generated by our compatibility analysis (§2.5.1). 1,102 classes already have RTTI and do not need modification since already protected by LLVM-CFI. Coping with all the issues at once is impracticable. Therefore, we adopt Property 3 to incrementally include compatible classes. For example, the team in charge of V8 [45] could apply type++ to Chromium instrumenting classes used in V8 only. This leaves the remaining classes unmodified by type++ and allows for a gradual deployment of type++ while reducing the risk of type confusion.

We develop a semi-automatic "delta-debugging" system that iteratively includes classes to Property 3, compiles, and validates Chromium. We additionally include a caching system to speed up Chromium compilation from 18 down to 6 hours on average, which remains the main bottleneck. Despite a lack of familiarity with the codebase, in roughly eight months, we manage to compile and run 1,928 of the 2,099 classes (91.8%) required by Property 2. These classes are linked to 2,428 warnings, leaving 911 warnings of Property 2 (27.3%) to further manual analysis. By considering all the classes already equipped with RTTI (34.4%) and those included with type++ (60.2%), we observe that type++ protects 1,928 classes (94.6%) involved in downcast operations in Chromium.

Chromium patching. From the compatibility analysis, we only modify 229 LoC out of 35 MLoC of Chromium C++ code base to support the above-mentioned 1,928 classes. Most of the changes involved assertions violated at compile time, *e.g.,* strict `sizeof` comparison with a scalar. Our patches disable these assertions. We plan to extend type++ with a special built-in macro which would make these assertions correct for standard C++ and type++ concomitantly.

The modifications regarding `protoc` were succinct and involved only three files for 28 insertions and 19 deletions in total.

Chromium performance evaluation. Our evaluation uses the average of 10 runs of JetStream2 [8], testing three Chromium configurations: baseline, LLVM-CFI, and type++ with support of 1,928 classes (94.6% of the total). Table 2.6 shows the final results. JetStream2 reported a score of 96.14 for baseline Chromium while LLVM-CFI suffers a 0.43% score reduction (95.73). type++ shows a slightly worse score of 94.80, indicating a 1.42% reduction of performance compared to the baseline. These results suggest that no major degradation of performance is caused by type++. These results are in line with the ones reported by the Chromium developers who observed that LLVM-CFI incurs less than 1% overhead when compared to the baseline [20]. The measured overhead does not affect the final user experience. type++ performance is on par with LLVM-CFI, and the additional 0.98% score reduction is caused by type++ protecting more than four times as many cast operations compared to LLVM-CFI (Table 2.6). Considering the severity of past type confusion vulnerabilities in Chromium, we argue that this extra overhead is reasonable. We are also confident that instrumenting the rest of the classes will not result in a vastly different performance since type++ already pays the cost by performing a failing type check for non-instrumented classes.

Comparison with other mitigations. The deployment of different type confusion protections has variable impacts in terms of compatibility and security guarantees. Here, we study the differences when applying type++, HexType, and LLVM-CFI—the current state-of-the-art. For HexType, we study its deployment over Firefox, which approximates the complexity of Chromium, thus comparing its reported results. LLVM-CFI is deployable on Chromium as part of the build system.

After porting 1,928 classes, type++ handles 741M down-to casts (out of 830M–89.7%). From code inspection, we conclude that the high coverage is caused by a narrow set of classes responsible for the majority of casts. HexType on Firefox protects a maximum of only 60% of the down-to casts. Similar to what was observed with SPEC CPU benchmarks, by protecting downcasting operations, we also stretch the protection of type++ over 69M unrelated casts, a form of cast that HexType cannot support. While the two browsers are not directly comparable, HexType in general protects fewer down-to casts and misses all unrelated casts compared to type++. The HexType authors mention that increasing their ratio of protected casts is impracticable as it would require complex code modification to Firefox or non-trivial adaptations of allocator tracking in HexType ([59]–Sec 5.1). In comparison, type++ already handles similar pool allocators natively (*e.g.,* Blender in SPEC CPU2017). On Chromium, type++ only misses casts of some classes due to the required engineering efforts to handle a project of this size. We further observe that HexType's implementation suffers from false positives, *i.e.,* type confusion that do not link to real bugs [97]. We reached out to the authors, who confirmed our observation. In certain cases, HexType suffers from stale metadata when the heap object lifecycle is not correctly handled (see

Table 2.6 – We compare Chromium baseline against LLVM-CFI and type++ with support of 94.6% of the necessary classes for Property 2, using the JetStream2 benchmark. A higher score is better. The third column is the number of casts protected. The last column is the percentage of all classes with RTTI, *i.e.,* including the polymorphic classes from the baseline.

Chromium	Perf. Score	(%)	Cast	Classes
baseline	96.14	-	-	34.4%
LLVM-CFI	95.73	0.43%	427M	34.4%
type++	94.80	1.42%	811M	94.6%

§2.1.3 for details), thus leading to false positive type confusions. The authors confirm that they do not delete old metadata to reduce runtime overhead. Conversely, for type++, every object embeds RTTI inline, thus carrying correct metadata throughout the object's lifecycle, and eradicating false positives by design. When evaluating LLVM-CFI on Chromium, we observe it detects only 347M (41.8% of the total) down-to casts. These numbers are in line with HexType when deployed on Firefox, and show the large attack surface left by previous works.

Attack surface. Figure 2.1 describes the attack surface when considering baseline, LLVM-CFI, and type++. From our analysis, there are 3,201 classes involved in down-to casts, 1,102 of them are already polymorphic, and 2,099 are non-polymorphic (POD). In the baseline, no classes are protected. Using LLVM-CFI cast protection, all the polymorphic classes become protected, *i.e.,* 34.4% of the total. The introduction of type++ pushes the tally to 94.6% of the total, almost three times more than LLVM-CFI.

Takeaway. Compared to LLVM-CFI, type++ manages to protect almost twice the amount of casts. Moreover, type++ reports no false positives and covers 69M unrelated casts, which would not be verified by HexType. After porting 1,928 classes, type++ already provides higher security guarantees than LLVM-CFI and HexType with minimal overhead.

2.6 Discussion

We discuss some potential challenges when porting source code to the type++ dialect.

Threats to type++ adoption. Previous dialects like CCured [92] and Ironclad [27], while pushing C and C++ in the right direction, failed to gain traction as a drop-in replacement. type++ proposes less drastic changes that are easier to adopt. Nonetheless, changing the language/dialect of a project may encounter resistance. We hope and will push for the inclusion of type++ into the main C++ compilers, initially as an alternative dialect, to ease accessibility to type++ and ensure future support beyond the effort of its original authors. Another likely source of opposition is the necessary integration into a wider software ecosystem, in particular with shared libraries. We

advise developers to favor Property 3 to ease adoption. However, this implies maintenance of the allow-list whenever dependencies change.

Following our experiments, we identified three patterns whose occurrence in a project make the adoption of type++ challenging. First, if a large subset of classes is shared between C and C++ code, the effort to synchronize the data layout is substantial (§2.6). Second, instability might arise if the project relies on undefined behavior (*e.g.,* fiddling with the LSB of pointers). Finally, the type field in objects should be protected against arbitrary writes, as laid out in our threat model (§2.2). This requirement is not trivial and likely comes with additional performance and memory overhead. Nonetheless, we consider these additional security guarantees a worthy trade-off between performance and security.

type++ aims to remain close to C++. New features are unlikely to cause new incompatibilities as the language moves towards type and memory safety as advocated by type++. From our experience, the effort of porting a codebase to type++ correlates negatively with how modern the codebase is. Codebases that expanded from C projects with little modernization take longer to port. As time passes, we argue that new projects will naturally follow type++ properties.

Interaction with C code. Projects may combine C and C++ code (*e.g.,* Blender, OMNeT++, Chromium). When compiling C code, the compiler must consider that some structures contain RTTI metadata, otherwise, the C code might misuse objects coming from C++ functions. To solve this problem, we have to synchronize the data layout across C and C++ modules. We envision three approaches: (i) generate new headers containing structures with an additional `vptr` only for C modules, (ii) modify the structures' data layout definition in LLVM, or (iii) add a Clang plugin that manipulates the C AST generation and adds a `vptr` to the instrumented classes. As cross-language interactions are rare, we opted to modify the structures' data layouts, *i.e.,* option (i). Note that the compiler can either change these types automatically or emit a warning for the developer.

Legacy code libraries. When interacting with legacy pre-compiled libraries, we cannot assume that their code is aware of the class modification. Here, we have two options: (i) we automatically infer the shared structures between legacy and type++ code and apply Property 3, ensuring equal data layouts for them in both code sections; (ii) we rewrite the legacy binary to adjust the field offsets. Neither option is perfect. As a fallback, we allow the programmer to manually specify classes shared with legacy code for Property 3.

Esoteric memory allocations. Some programs allocate contiguous memory regions that contain many objects in sequence. Our compiler automatically infers which objects the program is allocating and adds their constructors accordingly (see §2.4.3). To automatically locate unconventional allocators, an analysis can identify allocators that (i) cast to a specific known class, and (ii) the allocated size does not match the class size; *e.g.,* `A* a != (A*)malloc(sizeof(A) +`

`len`). As a sanity check, type++ separately reports any cast checks where the vtable pointer was uninitialized, highlighting the presence of unhandled allocators.

Functionality guarantee. It is key to maintain functionality when porting code from C++ to type++. If the changes mandated by type++ are not enforced, under specific conditions, the execution may deviate from the intended one. For example, in Listing 2.3, the comparison at Line 28 switches from `false` to `true`, changing the program's control flow. From our experience, patterns resulting in functionality difference remain rare; they are caused by unhandled C/C++ interactions as detailed above. To prevent such issues, the developer should investigate warnings reported by type++. Having a comprehensive test suite adds extra confidence but does not replace a proper investigation of the conflicting patterns.

Usage as a mitigation. Due to its low runtime overhead, type++ can be deployed as a mitigation in production to detect and protect against type confusion attacks. To this end, developers should ensure that the latest mitigations against memory safety vulnerabilities (*e.g.,* CFI) are deployed. This reduces the risk of attacks outside our threat model (§2.2) from modifying the type expected in the check or from skipping the type check completely. Additionally, type++ must be configured so that the program aborts when type++ detects a type confusion.

Supporting unrelated casts. Since Property 1 requires all classes to contain RTTI, type++ could enable type checks on all unrelated casts. This would guarantee full type safety in regards to all possible type confusions. This has a higher impact on performance as more type checks are performed due to the ubiquity of unrelated casts. Additionally, the porting effort increases as more classes need to be instrumented for Property 2. Finally, these classes are more likely to exhibit code smell since they already rely on the ill-advised `reinterpret_cast` operator [117]. We leave supporting unrelated casts for classes not involved in any downcast as future work.

Type confusions in non-C++ code. With Property 1, type++ guarantees the absence of downcasting type confusion in C++ code but cannot protect code in another language. Type confusion errors may still occur in non-C++ code, particularly C. Additionally, in the case of JavaScript engines, multiple type confusion vulnerabilities were reported in JavaScript JIT-generated code due to incorrect type tracking. We consider these issues orthogonal to type++ efforts.

2.7 Related work

type++ is a dialect that mitigates type confusion during typecasting operations. The literature already has C/C++ dialects addressing such problems, *e.g.,* Cyclone [61] is a type-safe dialect for C extending standard C with a set of protections that inspired other programming languages such as Rust [65] and Project Verona [87]. Likewise, Necula & al. introduced CCured [92], another C type-safe dialect that is similarly incompatible with the C++ specification. On the

contrary, type++ is explicitly designed to overcome C++ typecasting limitations. DeLozier et al. introduced Ironclad C++ [27], which enforces type safety in C++ programs. However, Ironclad C++ relies on the developers to manually adapt all classes to make them compatible with `dynamic_cast`. More recently, UNCONTAINED [66] looked for incorrect casts of C structure embeddings in the Linux kernel. Another approach for type safety relies on checkers to validate RTTI information at runtime. For instance, LLVM-CFI [120] and UBSan [22] perform type checks at runtime for polymorphic objects involved in unsafe casts. However, both tools only deal with polymorphic classes while ignoring Plain Old Data objects. Conversely, type++ validates both non-polymorphic and polymorphic classes. Moreover, UBSan requires heavy code modifications hindering deployment. Alternative approaches against type confusion use checks based on disjoint metadata structures [49, 59, 71, 95]. This introduces a large overhead and suffers from a high rate of false positive. On the contrary, type++ blocks unsafe casting efficiently and effectively by design, as evaluated in §2.5.2. Concurrent work [135] has looked at reducing the overhead by not adding type checks if the developer already implemented their own. This approach is orthogonal to type++ and hints at further possible performance improvements. Orthogonal efforts have looked at reducing the cost of the `dynamic_cast` type verification [39]. As type++ reuse this type verification implementation, these improvements would show an even greater performance benefit once deployed on top of type++.

2.8 type++ summary

We introduced type++, a new C++ dialect that explicitly assigns RTTI to all classes in a program. type++ allows fast runtime type checks thus overcoming runtime overhead, low coverage, and imprecision of previous works.

Our study on the effort to port standard C++ programs to type++ shows that our dialect requires changing only 0.16% of a program LoC in the worst case. Over SPEC CPU2006 and CPU2017, type++ incurs a negligible performance overhead (*i.e.,* 0.94%—two orders of magnitude faster than HexType) while protecting all the down-to cast operations (unlike previous works that are limited to only a subset). We find 122 type confusion errors in SPEC CPU2006 and CPU2017, 14 of them otherwise unobserved. Finally, evaluating type++ on Chromium results in an acceptable overhead (1.42%). All our findings, code, the material to replicate our experiments, and technical reports describing the type confusions found are publicly released.

Chapter 3

Sourcerer: channeling the `void`

> The more I learn, the more I realize how much I don't know.
>
> Aristotle

C++ offers high performance and flexibility but lacks strong type and memory safety guarantees, making it prone to type confusion (type confusion) vulnerabilities. These vulnerabilities, discussed in detail in Chapter 2, occur when objects are misinterpreted as incompatible types, leading to undefined behavior and potential exploitation. While existing solutions like LLVM-CFI and type++ mitigate some type confusion cases, they fall short of providing complete coverage, particularly for unrelated casts—explicit casts outside a class hierarchy. For example, type++ leaves 7,151M casts unprotected—(45.67% of all unrelated casts across both SPEC CPU2006 and CPU2017). In contrast, EffectiveSan [30] addresses all cast operations by encoding type information as low-fat pointers but requires heavy-weight modifications of all allocators and checks the type at each pointer dereference (which, compared to casts, is a very frequent operation), causing an unnecessarily high runtime overhead of 49%. *Reliably detecting all type confusions with a low runtime cost remains, therefore, an open challenge.* Additionally, after deep inspection, however, we observe that type++ only partially supports unions due to design limitations in the RTTI initialization.

In this chapter, we propose Sourcerer, a novel sanitizer capable of detecting *all* type confusions by enforcing a runtime type check at every explicit cast operation. The novelty of Sourcerer lies in embedding inline Runtime Type Information (RTTI) into all object types involved in unrelated casts, overcoming the restriction to derived casts that limits state-of-the-art solutions, *e.g.,* type++ and LLVM-CFI. To achieve this goal, Sourcerer introduces *RTTIInit*, a new C++ object initializer responsible solely of setting inline RTTI. With the introduction of RTTIInit, Sourcerer alleviates porting issues and overcomes the limitations of previous dialects, which forcibly injected default constructors and conflicted with the C++ standard. Furthermore, the introduction of RTTIInit solves the type initialization at union activation, another unsupported feature in existing dialects. As a result, Sourcerer performs runtime type validation for each cast from a non-generic types (*i.e.,* developer-defined types) and generic pointers (*i.e.,* `void*` or integral types), finally unlocking *full type testing* in C++ programs.

Sourcerer's pipeline consists of two stages. First, a static analysis identifies all classes involved in cast operations and helps the developer to adhere to the dialect. Then, our compiler generates an executable with the initializer responsible for setting the inline type information and the necessary type checks. We evaluate Sourcerer on the SPEC CPU2006 [50] and SPEC CPU2017 [13] benchmarks[1] and conduct a fuzzing case study on a leading C++ project, OpenCV [12], showcasing the ability to use Sourcerer to find illegal unrelated casts. Overall, Sourcerer detects all cast operations in the SPEC CPU benchmarks and incurs, on average, only a 5.14% performance overhead. Supporting objects involved in unrelated casts requires additional code changes. We deem the necessary adaptation of 453 out of 2,040K LoC reasonable with respect to the $4.95\times$ more classes instrumented compared to the instrumentation of only derived casts. In our evaluation, we conduct an ablation study to understand the root cause of the observed overhead. Our study concludes that the Application Binary Interface (ABI) change is the main cause of overhead highlighting the possibility of drastically improving performances by modernizing specific casts. In terms of security impact, Sourcerer identifies 152 type confusions in the SPEC CPU2006 and CPU2017 benchmarks. Finally, we conduct a fuzzing campaign in which we deploy our sanitizer on OpenCV [12], discovering six unrelated type confusion bugs, all missed by the state-of-the-art competitors.

Overall, the main contributions of Sourcerer can be summarized as follows:

- Quantifying *unrelated* casts so far hidden from state-of-the-art sanitizers.
- Proposing an extension to the type++ dialect to check *all* cast operations.
- A new initializer, RTTIInit, that sets object's type information, increasing the compatibility with the C++ standard and reducing the runtime cost.
- A thorough evaluation of Sourcerer against the state-of-the-art.
- A case study showcasing how Sourcerer can be used to detect vulnerabilities in real-world software through fuzzing campaigns on well-tested software.

We release the source code of Sourcerer and the documentation to replicate our experiments as open-source at `github.com/HexHive/Sourcerer`.

3.1 Background

In this section, we extend on the knowledge of Chapter 2 and distinguish more clearly unrelated from derived casts. Additionally, we introduce highlight the limitations of type++.

[1] When using SPEC CPU, we refer to the SPEC CPU2006 and CPU2017 benchmarks.

```
1  class Unrelated {char w;};
2  class Base {
3  public:
4    int x;
5    virtual ~Base()=default;
6  };
7  class Drvd:public Base{
8    int y[3];
9  };
10 class Sibling:public Base{
11   double z;
12 };
13 int main() {
14   Unrelated* u = new Unrelated();
15   Drvd* d = new Drvd();
16   Sibling* s = new Sibling();
17   // Safe casts: no need for verification //
18   Base* b = static_cast<Base*>(d);
19   Drvd* d2 = dynamic_cast<Drvd*>(b);
20   if(d2 == nullptr) return 1; //if cast fails
21   Base* b2 = static_cast<Base*>(s);
22   void* v = d; // Implicit generic cast
23   // ------------ Derived cast ---------- //
24   d2 = static_cast<Drvd*>(b2); // Illegal
25   d2 = reinterpret_cast<Drvd*>(b);
26   // ----------- Unrelated cast --------- //
27   d2 = static_cast<Drvd*>(v);
28   d2 = reinterpret_cast<Drvd*>(s);// Illegal
29   Unrelated* u2 = (Unrelated*) d; // Illegal
30   return 0;
31 }
```

Listing 3.1 – Examples of derived (lines 24 & 25) and unrelated casts (lines 27–29). We omit statistics on safe casts (upcast and `dynamic_cast`, lines 18 & 19).

3.1.1 C++ casting

Casting—adjusting the type of objects—allows for flexible and generic code. While implicit conversions (*e.g.*, conversions defined by the C++ standard like `char` to `int`) or the developer-defined ones are safe, explicit casts pose the risk of undefined behavior. Specifically, unsafe casts have two origins: *derived or down casts* occur when the resulting type inherits from the source class while *unrelated casts* encompass conversions between two types not related by inheritance (*e.g.*, `void` or other generic pointer type).

Both unrelated and derived casts can be safe. For example, casting from `void*` is well-defined if the pointed memory was previously cast from the desired type (*e.g.*, Line 27 in Listing 3.1). Outside these cases, the result of the cast is undefined. For example, in Line 29 in Listing 3.1 a `Drvd` object pointer is illegally cast through `static_cast` to a `Unrelated` pointer. After such a cast, the program might try to access the `w` attribute of the `Unrelated` object which is not part of the `Drvd` objects. Such illegal casts are referred to as *type confusions* and might lead to memory corruption. To avoid such errors, C++ developers need to maintain a mental model of the actual object types. To do a type conversion, the C++ standard mandates the use of one of the following explicit cast operators if the conversion is not inherently safe—*e.g.*, neither implicit nor defined by the developer.

3.1.2 type++ Limitations

Research for detecting type confusions has evolved with diverse mechanisms being explored. In the Chapter 2, we proposed a new C++ dialect mitigating, by design, type confusions in derived casts. By extending all derived cast objects with Runtime Type Information (RTTI), type++ can protect each derived cast, maintaining their type safety throughout the program execution. Adding RTTI into these objects changes the Application Binary Interface (ABI) resulting in some (limited) porting effort. To set the RTTI, their implementation artificially injects default constructors to all classes involved in down-to casts. This solution conflicts with some C++ idioms like if it is marked as `deleted` or when constructor calls are actively avoided (§3.4.5).

Crucially, type++ stops short of being *completely type safe* as it only partially protects unrelated casts—slightly less than half in the case of the SPEC CPU benchmarks. Moreover, instrumenting only a subset of cast classes breaks the implied compatibility between some classes resulting in unnecessary porting effort. We hypothesize that the reliance on the constructor to set RTTI is the root cause of these porting issues.

3.2 Threat Model

Sourcerer is a sanitizer for detecting *all*, derived and unrelated, type confusions in C++. In particular, when a type confusion bug is triggered during testing, we expect Sourcerer to detect the type safety violation. We assume a correct implementation of our compiler and the program to be correctly ported to the type++ dialect. Specifically, Sourcerer is not designed for an adversarial scenario due to possible false negatives if an attacker can leak and set type information. We refer to §3.8 for a thorough discussion. In summary, our threat model aligns with the ones from previous type confusion sanitizers [59].

3.3 Challenges

Upon careful evaluation of related works, we identified several unsolved challenges for type confusion sanitizers. First, the state-of-the-art, *e.g.,* type++ and HexType, offer incomplete type safety as they miss most unrelated casts. Only EffectiveSan offers theoretically full type safety but at an unnecessary high runtime cost, creating the second challenge we aim to address. Lastly, we identified only partial support for unions in type++ and EffectiveSan while Sourcerer provides complete support.

- **Unrelated casts**: To detect all type safety violation, a sanitizer needs to cover *all* cast operations, including unrelated casts.

- **Performance overhead**: For a sanitizer to be widely adopted, it should have a minimal performance overhead.

- **Union support**: C++ unions allow different types to refer to the same memory. This is a problem for type confusion sanitizers as they need to track the union type in memory.

3.4 Sourcerer's Design

In this section, we introduce the core concepts allowing Sourcerer to address these challenges. First, we lay out which classes need type information to truly check *all* casts. Then, we describe *RTTIInit*, an optimized inline type information initializer reducing type++ dialect divergence and lowering the performance overhead by $3\times$ in comparison with the other complete type confusion sanitizer, EffectiveSan. Additionally, we explain the key properties of RTTIInit allowing the support of unions. Finally, we list the idioms unsupported by earlier work that Sourcerer handles, like templates for EffectiveSan.

3.4.1 Classes to Instrument

Sourcerer's core contribution is to check all casts. Specifically, Sourcerer instruments all the classes involved in any derived or unrelated casts, thus extending Property 2 laid out in §2.3. Formally and in line with the properties defined in Chapter 2, we refer to this specialization as *Explicit Runtime Types For All Casts*:

Property 4 (Explicit Runtime Types For All Casts.) *Given all classes CS of a program P, Sourcerer associates a unique type T to each class $A \in CS$ if A is either the destination or the type of the source object of an explicit cast.*

This new property allows for the verification of all type casts while keeping the number of classes to instrument at a minimum. Due to the ubiquity of casts from `void*`, the sanitizer needs to instrument an increased number of classes—1,043 additional ones in the SPEC CPU benchmarks, a $5\times$ increase compared to previous works [6] targeting only derived casts. The key insight to implement this property lays in RTTIInit, that allows transparent type information initialization in objects without interfering with the C++ constructors.

```
1 mov     $0x1,%edi
2 call    47340 <malloc@plt>
```

```
1 mov     $0x10,%edi
2 call    47340 <malloc@plt>
3 call    46f80 <_ZN9RTTIInitUnrelated>
4 ...
5 <_ZN9RTTIInitUnrelated>:
6 lea     0x1ad9(%rip),%rax #vtable address
7 mov     %rax,(%rdi)
8 ret
```

Listing 3.2 – Assembly code of the `malloc`-ation of an `Unrelated` object from Listing 3.1 without and with RTTIInit.

3.4.2 RTTIInit

The ability to type check an object at runtime depends on the presence of type information. Typically, approaches using external metadata to track object types struggle to follow all object lifetime events (*e.g.,* copy). Inline metadata, as pioneered by LLVM-CFI and type++, is more robust as the type information is stored in the object and not disjoint from the object such as for TypeSan [49] and HexType [59]. The type information will be carried in the different lifetime events. Therefore, only the object creation requires careful handling to ensure the type information is correctly initialized. Typically in C++, object creation happens through `new` or direct assignment, which both call the object constructor which also sets RTTI when required. The type++ implementation followed this approach, by forcing constructor calls for object creation not relying on any initialization but only allocation, *e.g.,* `malloc`. This approach breaks different idioms in the C++ standard like explicitly `deleted` constructors or `const` objects where initialization should occur only once. To avoid these issues, Sourcerer introduces a new initializer, *RTTIInit*, uniquely focused on setting RTTI as exemplified in Listing 3.2. By interacting only with the RTTI field, Sourcerer avoids incompatibilities with `const` qualifiers and minimizes the performance cost of setting the RTTI. Additionally, RTTIInit does not interfere with the remaining object content, allowing for complete support of unions as detailed in §3.4.3. Lastly, our initializer, as a new language feature, does not conflict with existing constructors or the lack thereof which was a limitation of type++ that complicated its deployment.

3.4.3 Support for Unions

Unions, in C++, use the same memory to store objects of different types, allowing, however, only a single type to be active at a time. The union switches type when a member is activated, either through a direct assignment (Line 19 in Listing 3.3) or by setting a field of a union member. As the core property of Sourcerer is to maintain inline type information throughout the program execution, Sourcerer needs to ensure that, upon activation, the type information is correctly updated in memory. Direct assignments do not require further handling as the incoming object's RTTI is already set. As a field assignment can occur when the object is already activated, calling

```
1  using namespace std;
2  union Union {
3      int i;
4      char c[2];
5  };
6  struct X { Union u;};
7  union BiggerUnion {
8      X x;
9      OtherClass c;
10 };
12 int main() {
13     BiggerUnion bu; // No member active
14     bu.x.u.i = 65; // bu.x and bu.x.u.i active
15     // u should only be accessed as integer.
16     cout << bu.x.u.c[0] <<endl; // UB
17     bu.x.u.c[0] = 0x42; // u is active as char[]
18     bu.x.u.c[1] = 0x41;
19     bu.c = OtherClass(); // Direct assignment
20     return 0;
21 }
```

Listing 3.3 – Abbreviated snippet showing valid and invalid union member accesses. The assignment at line 14 activates `bu.x` and `u.i`. Access to `u` through a non-active member leads to undefined behavior (*e.g.,* line 16) [24]. Sourcerer's RTTIInit is called at each access while type++ misses instrumenting line 17 as calling the constructor would overwrite `bu.x` and `u.c`.

a constructor would overwrite the stored object content (*e.g.,* Line 17 in Listing 3.3). Sourcerer, however, calls RTTIInit at each field assignment, which sets the RTTI without modifying the remaining object content.

3.4.4 Dialect Simplification

Sourcerer relaxes two dialect requirements introduced by type++. First, the type++ compiler requires a default constructor to be defined for each instrumented class and has to relax the `deleted` attribute in case the default constructor is defined as such. Sourcerer's instrumentation, on the other hand, does not use constructors and retains the intention of the developer. Sourcerer, additionally, remove type++ changes for `const` variables initialization. Relying on a constructor to set RTTI imposes a second initialization step, breaking the single initialization requirement of `const` variables. Conversely, as RTTIInit is not counted as an actual initialization step, it averts any limitation for `const`.

3.4.5 Unsupported Idioms in Earlier Work

While deploying Sourcerer, we encountered a C++ idiom that type++ could not support. Specifically, libc++, the LLVM C++ standard library, explicitly avoids calling the object constructor when allocating a tree node to allow for constructor homing [78], an optimization reducing the amount of emitted debug information. Instead, they allocate a `char` array and then cast it to the desired type as shown in Listing 3.4. type++ either breaks the optimization or cannot set the RTTI, disregarding type safety. Sourcerer, on the other hand, can instrument this peculiar allocation pattern without breaking the optimization.

```
1 template <class _Tp>
2 struct __list_node : public __list_node_base<_Tp> {
3   // Starting the lifetime of nodes without initializing in order to be allocator-aware.
4 private:
5   _ALIGNAS_TYPE(_Tp) char buf[sizeof(_Tp)];
6
7 public:
8   _Tp& __get_value() {
9     return *__launder(reinterpret_cast<_Tp*>(&buf));
10  }
```

Listing 3.4 – Challenging idioms in the libc++ 19.0.0 `list` implementation. A node is allocated through a `char` array later cast to the desired type. The constructor is later called via `construct_at`. type++ calls the constructor at line 5 thereby unfortunately disabling the constructor homing optimization [78]. Sourcerer, on the other hand, can either allow-list the cast at line 9 or call the RTTI initializer at line 5 without breaking the optimization.

When evaluating EffectiveSan, we identified two unsupported idioms. First, Custom Memory Allocator (CMA) need to be replaced, incurring many LoC modifications. In comparison, Sourcerer only requires a list of CMAs and handles their instrumentation automatically. Secondly, similarly to the type++ authors, we encountered a false positive due to incomplete handling of C++ templates, an issue not faced by Sourcerer.

3.5 Implementation

Sourcerer needs to add inline metadata to the necessary classes and instrumentation to verify for all casts. We implement the Sourcerer prototype on top of the modular compiler toolchain, LLVM 19.0.0 [68]. Specifically, we port the class collection, custom allocator logic, as well as the warning analysis from type++ to LLVM 19.0.0. For RTTIInit, we copy the logic of the default constructor, but trim it down to set only type information *i.e.,* the vtable and the RTTI. The resulting ABI change is similar to the one in type++—any function interacting directly with the object size might be problematic. For verification, we rely on LLVM optimized type checks. Moreover, compared to type++, we add support for multiple allocator edge cases such as zero-size allocation and frees through `realloc`. Overall, our implementation totals 9K LoC and consists of two compilation passes, first to gather the types and then to instrument them. We open-source Sourcerer and the evaluation at github.com/HexHive/Sourcerer.

3.6 Evaluation

Sourcerer's evaluation targets the following research questions:

- **RQ1:** What extra efforts are required to check all unrelated casts?
- **RQ2:** What is Sourcerer runtime overhead compared to the state-of-the-art?
- **RQ3:** Which source causes the performance overhead introduced by Sourcerer?
- **RQ4:** How effective is Sourcerer at detecting type confusion vulnerabilities?
- **RQ5:** How does Sourcerer perform in a real-world bug-hunting scenario?

Experimental setup. Our evaluation runs in Ubuntu 20.04 Docker containers on a server with two Xeon E5-2680v4 @ 2.4GHz and 256GB of RAM.

Evaluation targets. Sourcerer's evaluation is twofold: first, we compare it to the state-of-the-art, then we demonstrate Sourcerer effectiveness on current large-scale projects. As such, we evaluate Sourcerer on the SPEC CPU2006 [50] and SPEC CPU2017 [13] benchmarks as they are the common benchmarks across the state-of-the-art. For both, we select all the C++ programs and compile them with `-O2` optimization level. Overall, we evaluate Sourcerer on 2,040K LoC lines of code across the 16 programs of the SPEC CPU suites. Additionally, we conduct a fuzzing campaign against OpenCV [12] (commit `796adf`), the state-of-the-art computer vision library, showcasing the ability to use Sourcerer to find illegal unrelated casts. We choose OpenCV as it is a large, 1.3M LoC, and popular C++ project with a fuzzing setup readily available from OSS-Fuzz [109].

State-of-the-art competitors. We compare Sourcerer against a representative set of the state-of-the-art type confusion sanitizers. Specifically, we choose type++ [6] as it is the most recent type confusion protection and the cast checker of LLVM-CFI [120] due to its use in industry. Finally, we report numbers from EffectiveSan [30] as it is the only other tool claiming to protect unrelated and derived casts. Despite our efforts, we were unable to run EffectiveSan as the cast checking configuration is neither present in the source code nor in the documentation. Lastly, EffectiveSan checks types at pointer access, and, therefore, does not report the number of cast operations protected. This discrepancy makes a quantitative security comparison of Sourcerer and EffectiveSan meaningless.

For each tool, we report the performance overhead, averaged across five runs, and compare it to the corresponding LLVM vanilla version—13.0.1 for type++, and 19.0.0 for LLVM-CFI and Sourcerer. To assess the effectiveness of Sourcerer, we report the number of runtime cast operations checked, similarly to type++. Every configuration is run with Link-Time Optimization (LTO) enabled as required by LLVM-CFI cast checking.

3.6.1 Porting Effort

Sourcerer follows the type++ dialect specification, and, therefore, requires similar porting efforts to translate C++ code into its dialect. Starting from type++'s open-source patches, we address the additional warnings caused by the classes involved in unrelated casts. As a first metric, we report the number of extra classes that need to be instrumented to check unrelated casts. For these extra classes, we break down the kind of unrelated cast causing their instrumentation in the column *Unrelated* in Table 3.1. On average, Sourcerer instruments and monitors $4.95\times$ more classes than type++ as our checks stretch beyond derived casts. Some programs like POV-Ray have few classes, *e.g.,* 12, involved in derived casts but extensively use unrelated casts with 161 classes cast. SoPlex experiences a less dramatic increase, *e.g.,* $3\times$ more classes, with most classes cast as part of libc++ data structures headers (*e.g.,* vector). This reduces the overall effort as porting the library is amortized across the different programs. In Table 3.1, we report the classes requiring explicit instrumentation. The total number of types instrumented is a superset as some classes inherit the instrumentation and templates are counted once and not per specialization.

The actual porting effort caused by the new classes is small. For the SPECCPU benchmarks, we only modify 120 LoC on top of the type++ patches, for a total of 453 LoC patched. The new changes are relatively minor, for example, an initialization procedure (*e.g.,* `placement_new`) in NAMD 2017. Around 20% of the LoC changed are fixes for type confusions. They are necessary because the ABI changes do not always allow Sourcerer to recover from the subsequent memory corruption. For example, in Povray 2017, a parent object is created by `malloc`-ing enough memory but storing the returned pointer in a variable of a larger child type as exemplified in Listing 3.5. As Sourcerer identifies the returned memory as a child object, it calls RTTIInit which set information out of the allocated bounds, leading to memory corruption. The biggest changes were in Blender, which interacts heavily with C code resulting in two challenges. First, as some structs are defined in headers included in both C and C++ code, we had to mimic the presence of RTTI information in the C code to ensure a compatible ABI. The second issue arises when, in C code, an instrumented C++ object is cast to a pure C struct. As Sourcerer only instruments C++ code, the cast is not checked nor does the destination type expect the RTTI field. We modified the C struct to be aware of the presence of type information. Moreover, instrumenting more classes showed some unexpected benefits as we could remove a patch from type++ for SoPlex as layout similarity is restored between two types involved in an unrelated type confusion.

EffectiveSan preserves the C++ ABI but struggles with custom allocators. In their artifact, the changes necessary for SPEC CPU2006 totaled a non-trivial 297 LoC. In contrast, Sourcerer needs to modify 453 LoC while incurring, in the worst case, only a third of EffectiveSan's average runtime overhead.

Overall, we conclude that the porting efforts for Sourcerer are reasonable and in line with the efforts necessary for similar tools.

Table 3.1 – Breakdown of the number of classes instrumented by Sourcerer as well as the number of RTTI initialization. *Total* shows the number of classes that Sourcerer instruments which is broken down into the cause of the instrumentation. *Derived* indicates classes involved in derived cast and, therefore, already instrumented by type++. Then, we report the additional classes involved in unrelated casts either from a specific type (*i.e., class*) or from generic pointers (*e.g.,* `void*`). The last two columns indicate the number of instrumented objects and the percentage which are initialized through RTTIInit.

	Program	LoC	Instrumented classes				# RTTI Init.	% Through RTTIInit
			Total	Derived	Unrelated			
					class	`void*`		
SPEC CPU2006	NAMD	4K	27	9	4	14	460K	100.0
	deal.II	95K	63	12	4	47	15,875M	98.32
	SoPlex	28K	39	13	6	20	726M	5.91
	POV-Ray	79K	161	12	23	126	6,243M	99.97
	OMNeT++	27K	53	13	4	36	2M	65.82
	Astar	4K	31	9	4	18	6,024M	91.97
	Xalan-C++	264K	149	46	5	98	1,407M	99.86
SPEC CPU2017	cactuBSSN	63K	31	11	5	15	334K	81.19
	NAMD	6K	26	9	4	13	0	-
	Parest	359K	120	26	4	90	20,384M	98.53
	POV-Ray	80K	161	12	23	126	25,179M	100.0
	Blender	616K	57	12	20	25	70M	74.3
	OMNeT++	86K	125	14	5	106	3M	38.93
	Xalan-C++	291K	203	46	7	150	35,276M	99.61
	Deep Sjeng	7K	27	9	4	14	15M	0.0
	Leela	31K	34	11	4	19	4,491M	99.53
	Total	2,040K	1307	264	126	917	-	-

3.6.2 Performance Overhead

Since speed is key for automatic testing, sanitizers should incur a limited performance overhead (§3.3). Below, we quantify the performance cost of deploying Sourcerer on the SPEC CPU benchmarks and compare it to the state-of-the-art.

Table 3.2 details the performance overhead of the different tools in the columns marked "%". Each value is the average performance overhead compared to a binary compiled with vanilla Clang—version 13.0.1 for type++, 19.0.0 for both LLVM-CFI and Sourcerer. We observe a negligible standard deviation across the five runs. As expected, the additional classes instrumented increase the overhead of Sourcerer compared to LLVM-CFI and type++. While LLVM-CFI and type++ are mitigations, Sourcerer's higher overhead is outstanding for a sanitizer deployed in

Table 3.2 – Performance and security evaluation of Sourcerer compared to the state-of-the-art. Under "*%*" we report the performance overhead compared to the tool's baseline. Then, in the *Casts* column, we report how many unrelated casts were checked at runtime. The two Δ columns show the extra operations verified by Sourcerer on top of the competitors. The last two columns report the average and peak memory overhead of Sourcerer in terms of the working set size.

	Program	LLVM-CFI		type++		Sourcerer		Δ Casts		Memory	
		%	Casts	%	Casts	%	Casts	LLVM-CFI	type++	Avg.	Max.
SPEC CPU2006	NAMD	0.41	1	0.27	0	0.53	1	0	1	0.45	0.45
	deal.II	-1.15	649K	1.95	122M	4.33	128M	127M	5M	-2.17	1.95
	SoPlex	-0.01	206K	0.22	27M	12.64	30M	30M	4M	66.79	66.83
	POV-Ray	0.92	1M	1.60	1,342M	7.93	1,345M	1,344M	4M	10.39	10.39
	OMNeT++	3.42	3	0.67	270K	3.04	284M	284M	283M	1.43	1.43
	Astar	0.85	0	0.90	0	16.99	4M	4M	4M	104.89	82.03
	Xalan-C++	0.63	5K	-0.54	5K	-1.29	161K	156K	156K	9.28	2.85
SPEC CPU2017	cactuBSSN	0.47	0	-0.71	100	-0.21	21K	21K	21K	0.30	0.08
	NAMD	0.52	5	-0.68	0	0.71	0	-5	0	0.23	0.22
	Parest	0.36	24K	0.07	85M	2.46	106M	106M	21M	2.11	1.99
	POV-Ray	0.08	39K	1.73	5,370M	9.59	5,381M	5,381M	11M	10.98	10.96
	Blender	0.69	0	2.92	7M	9.62	7,643M	7,643M	7,636M	2.78	3.01
	OMNeT++	1.31	6M	0.34	6M	4.11	501M	495M	495M	2.21	2.12
	Xalan-C++	-0.10	4K	-0.24	198M	15.38	243M	243M	45M	4.94	4.43
	Deep Sjeng	-0.14	0	-1.09	0	0.45	1	1	1	16.37	16.37
	Leela	-0.53	0	-0.27	1K	-0.19	416K	416K	414K	15.69	5.69
	Avg./Total	0.26	8M	0.43	7,158M	5.14	15,666M	15,658M	8,507M	-	-

a testing environment. For example, UBSan [22] manifests slowdowns of up to ~1.7x. More precisely, Sourcerer incurs a 5.14% performance penalty but covers all casts in a program. When looking only at SPEC CPU2006, the target set of EffectiveSan, Sourcerer overhead is limited to 6.47% compared to the 49% overhead reported by EffectiveSan's authors. Consequently, Sourcerer reduces the runtime overhead for a complete sanitizer by a factor of seven. In fact, Sourcerer shows a similar overhead to HexType [59] but additionally checks unrelated casts and avoids false positives by design.

Looking at individual programs, we observe that the overhead is not uniform. As shown in Table 3.2, programs such as cactuBSSN, NAMD and Deep Sjeng experience virtually no overhead but also check the fewest casts. On the other hand, SoPlex and Xalan-C++ 2017 suffer from an overhead of around 15% due, in part, to caching being undermined by bigger objects on hot paths. To conclude, a complete type confusion sanitizer is practical if the checks occurs at cast time and not at the frequent dereference sites, as implemented in EffectiveSan.

3.6.3 Source of the Performance Overhead

In this section, we study the performance cost of Sourcerer instrumentation. In particular, we conduct an ablation study on the three following elements: the type checks, the RTTI initializer, and the ABI changes. First, we disable the type checks but leave the instrumentation intact. Then, we also remove the call to RTTIInit, leaving only the changes to the ABI in place. Each experiment is compared to the same vanilla Clang baseline. We present the results in Table 3.3.

Comparing the columns *Full* and *W/o type checks* shows that the verification cost can be important, *e.g.,* Blender. Upon closer inspection, Blender exhibits a high ratio of failing type checks which is a slow path. Indeed, in a testing environment, we expect the program to be terminated upon encountering a type confusion while in this evaluation we continue to assess the program performance. Fixing these type confusions would reduce the cost of the checks to a similar level as in Xalan-C++ 2006 which proceeds to more, but successful, type checks. Overall, the numerous optimizations implemented in LLVM type checks [79] allow for this limited performance cost.

Inspecting the column *ABI change only*, we observe that altering the object size negatively affects the caching behavior. Indeed, in Astar, we observe that the class `pointt` is instrumented by Sourcerer. As a single byte object, adding RTTI double its size, negatively impacting caching in the tight loops of the `flexarray::add` function. Reducing this overhead could be achieved by removing all casts of `pointt`, and, therefore, Sourcerer instrumentation. This would return Astar to the original caching performance and a minimal overhead. A similar issue is observed in SoPlex. The cost of the ABI change varies a lot across programs and use-cases. Compared to type++, this effect is magnified in Sourcerer by the additional classes instrumented.

Lastly, Table 3.3 allows us to estimate the overhead of Sourcerer's RTTIInit as it is the only change between the columns *W/o type checks* and *ABI change only*. Disabling RTTIInit does not lead to a significant change in performance, highlighting the effectiveness of Sourcerer's RTTI initialization design.

Overall, this study highlights the cost of the ABI change, which is strongly dependent on the program and its use-cases. Nonetheless, the average overhead of Sourcerer is lower than other sanitizers while detecting all type safety violations.

3.6.4 Security Effectiveness

Sourcerer is a sanitizer deigned to detect *all* type confusions. In this section, we compare the number of cast checks against similar tools and describe the type confusions that Sourcerer identified which were missed by previous works. From a theoretical point of view, both EffectiveSan and

```
1 class X {}; // Instrumented
2
3 class A {
4   int x; // A is instrumented
5 };
6
7 class B : public A {
8   int y;
9   X x; // Need to set x
  ↪  \acrshort{RTTI}
10 };
11
12 int main() {
13   A* a;
14   a=(B*)(malloc(sizeof(A)));
15   free(obj);
16   return 0;
17 }
```

Listing 3.5 – Simplified excerpt of a type confusion in POV-Ray 2017. Sourcerer calls RTTIInit at line 14 right after the call to `malloc`. Sourcerer assumes, from the cast, that the underlying object is of type `B` while the allocated size is only sufficient for an `A` object. The call to `A`'s RTTIInit will try to set `x`'s RTTI information (line 9), resulting in an out-of-bounds write.

Table 3.3 – Ablation study of Sourcerer performance overhead. The column `Full` lists the overhead of Sourcerer compared to the baseline. The third column shows the overhead once the type checks are disabled. For the last column, we additionally disable calls to RTTIInit, highlighting the cost of the ABI change. Comparing `Full` and `W/o type checks` shows the overhead of the cast validation. Finally, comparing the last two columns allows for an estimation of the overhead induced by RTTIInit.

	Program	Full	W/o type check	ABI change only
SPEC CPU2006	NAMD	0.09	0.15	-0.15
	deal.II	3.89	5.36	3.53
	SoPlex	12.88	12.87	8.80
	POV-Ray	8.58	5.20	5.41
	OMNeT++	6.28	1.37	0.05
	Astar	16.51	16.86	15.53
	Xalan-C++	-1.83	1.42	0.48
SPEC CPU2017	cactuBSSN	0.18	-1.01	-0.30
	NAMD	-0.15	-0.06	0.75
	Parest	1.77	2.32	2.07
	POV-Ray	9.73	5.68	5.95
	Blender	9.16	1.00	1.26
	OMNeT++	5.96	2.29	2.90
	Xalan-C++	14.64	13.97	15.34
	Deep Sjeng	0.15	0.06	0.16
	Leela	-0.50	-0.82	-0.51

Sourcerer provide complete coverage of all cast operations. We do not provide statistics about EffectiveSan checks as they do not happen at cast time but at every object dereference. However, as mentioned in §2.5, we observed false positives in EffectiveSan due to incompatibilities with templates.

Table 3.2 reports the number of casts checked by LLVM-CFI, type++, and Sourcerer. For each program, we list the number of unrelated casts verified during the benchmark execution. The difference between Sourcerer and both HexType and LLVM-CFI is listed in the two columns headed by Δ *Casts*, respectively. LLVM-CFI checks a mere 8M unrelated casts, less than 1% of all unrelated cast operations, while type++ already verifies slightly less than 50% due to classes being involved in both derived and unrelated casts. Sourcerer covers the remaining 50% of casts, reaching all 15.7B unrelated casts, highlighting the wide attack surface left unchecked by previous

research. The increase in verified casts is dominated by Blender, but almost all programs benefit from the additional type confusion detection capabilities offered by Sourcerer. NAMD 2017 is a special case where type++ and Sourcerer report fewer casts due the changes necessary for the dialect which removed the only cast triggered in the execution.

In regards with the security impact, Sourcerer discovered 152 type confusions in the SPEC CPU benchmarks—30 more than the state-of-the-art. Most of these type confusions expect classes to have identical layouts. Despite the lack of guarantee from the C++ standard, C++ compilers rarely optimize object layouts thus, reducing the risk associated with these type confusions. Nonetheless, relying on such undefined behaviors is unsafe despite its widespread use.

3.6.5 Sourcerer as a Sanitizer for Fuzzing Campaigns

In this section, we showcase the effectiveness of Sourcerer as a sanitizer by using our prototype in combination with AFL++[33]. We conduct the first fuzzing campaigns targeting specifically type confusions and compare its performance with a state-of-the-art memory sanitizer, AddressSanitizer (ASan) [106].

As target, we select OpenCV [12], the leading computer vision library, due to its ubiquity and the availability of fuzzing drivers as part of the OSS-Fuzz project [109]. We unleash AFL++, a state-of-the-art fuzzer, on the seven drivers and conduct five 24-hour fuzzing campaigns for both ASan and Sourcerer. We replay the inputs on a binary instrumented with SanCov [119] to have collision-free coverage and avoid different numbers of edges due to the instrumentation.

Thanks to Sourcerer's increased compatibility with the C++ standard, few changes are necessary to support the 1.34M C++ LoC of OpenCV. A peculiar issue is how the characteristics of matrices elements (*e.g.,* depth, size, and number of channels) are stored and later used to compute the offset between elements, shunning `offsetof`. As Sourcerer modifies this offset, matrices traversal results in RTTI being interpreted as matrix elements. Indeed, the type++ dialect is incompatible with hard-coded object sizes. More importantly, this is also less portable and needs support through `#ifdef` and header files. We leave out this code modernization as it involves modifications of the core components of the library. Instead, we use Sourcerer flexibility to disable the instrumentation of the five matrix element types—out of 668 classes involved in casts—trading a slight reduction of the findable type confusions for easier deployment of Sourcerer.

Figure 3.1 shows the performance of the three fuzz drivers. In shaded colors are the minimum and maximum values achieved across the five repetitions. The left graphs highlight that Sourcerer achieves a similar branch coverage to ASan, despite being hindered by type confusion crashes. On the right, the total number of executions of the fuzz drivers shows that Sourcerer instrumentation is more lightweight than ASan, allowing to test more inputs.

In terms of crashes, the two drivers, `imread_fuzzer` and `core_fuzzer`, highlighted in Figure 3.1, triggered three type confusion bugs. The crashes are caused by similar unrelated casts to a `PaletteEntry` object at three different code locations. Our testing found three additional type confusions in code assuming indistinguishability between an array of `type`, *e.g.,* `float`, and a `Vec<type>` objects, representing a vector. However, as the array is oblivious to the `Vec` RTTI field, the `reinterpret_cast` results in shifted values and incorrect RTTI. Sourcerer's instrumentation makes this type confusion apparent but blocks the program execution as the object is corrupted, hindering fuzzing progress. To highlight the capabilities of Sourcerer, we investigate if type++ can identify the errors. Since type++ does not instrument `PaletteEntry` and `Vec`, type++ was unable to detect these errors, further showcasing the effectiveness of Sourcerer.

Previous type confusion sanitizers never conducted fuzzing campaigns likely due to false positives (*e.g.,* HexType) or the expensive porting effort (*e.g.,* CMAs in EffectiveSan). Sourcerer, therefore, is the first to demonstrate the feasibility and effectiveness of fuzzing campaigns with a complete type confusion sanitizer.

3.7 Related Works

In the next paragraphs, we discuss relevant works for type confusion sanitizers.

Type confusion defenses. TypeSan [49] and HexType [59] check derived cast by tracking object types throughout their lifetime in an external data structure. The complexity of C++ lifetimes leads to prohibitive cost and a high rate of false positives [97]. EffectiveSan [30] encodes, through fat pointers, the type and bounds of an object. At each pointer dereference, they perform a bound check and a type check causing a high runtime cost. Multiple dialects exist for C++ to prevent by design certain classes of vulnerabilities. Ironclad C++ [27] banned unions and added type information to every object requiring large changes to the source code. More recently, type++ [6] (Chapter 2) proposed limited code changes to add RTTI to each object involved in a derived cast. Finally, UNCONTAINED [66] identifies derived type confusions in C containers, particularly in the Linux kernel.

Type check pruning. To avoid type confusions, developers implement their own type identifiers and checks. Recent works investigated disabling such checks when a sanitizer is deployed. In particular, Zhai & al. [135] automatically remove type checks redundant with HexType to improve performance. Orthogonally, HTADE [31] removes derived casts that are never dereferenced.

3.8 Discussion

In the following, we detail Sourcerer's limitations and possible extensions.

Custom allocator identification. Similarly to previous tools, Sourcerer tracks object lifetime and, therefore, their allocation. While `new` and direct initialization sets RTTI automatically, C style allocations (*e.g.,* `malloc`, `calloc`, and `realloc`) require to explicitly initialize the RTTI through RTTIInit. Additionally, Custom Memory Allocator (CMA) wrap the standard allocation functions to provide extra features like memory pool or allocation metadata (*e.g.,* ASan). To initialize RTTI, Sourcerer as other sanitizers, must be aware of these allocators and is, therefore, configurable through an allow-list. Orthogonally, CMAsan [52] recently automated CMAs identification in C++ projects.

Identification of Allocation Type. To correctly set type information after an explicit allocation, Sourcerer needs to know the allocated type. Assuming the presence of a cast to the desired type right after returning from the allocation has proven sufficient in our evaluation. A sounder static analysis would be able to follow allocation until their first actual assignment.

Reliance on Link-time optimization. Sourcerer relies on the type checks implemented by LLVM-CFI which leverages the LLVM `type.test` function [79]. To allow optimized checks, it leverages Link-Time Optimization (LTO) to optimize inheritance hierarchies. During libc++ instrumentation, we discovered that some casts were unchecked by LLVM-CFI as their LTO-visibility attribute is set to expose symbols to external compilation units, preventing LLVM from emitting type checks. Clang provides an option, `-fsanitize-cfi-cross-dso`, to check externally visible objects across Dynamic Shared Objects (DSO), at the cost of lower performance and extra compilation requirements (*e.g.,* position-independent code) [76]. We leave evaluating this option as future work.

Future Work. Accessing a union through a non-active member is undefined in C++ [55]. Practically, the illegal access is identical to a `reinterpret_cast`. To solve this issue, Ironclad C++ [27] banned unions in their dialect. Adding type information to the types used in unions would allow Sourcerer to check the validity of union access at the cost of additional porting effort and performance overhead. We leave verifying union access correctness as future work.

Sourcerer is a sanitizer and, therefore, is not suited for an adversarial scenario. Multiple improvements are necessary to mitigate type confusions. First, all source objects need to be typed, as otherwise, an attacker controlling the object content could forge RTTI values, resulting in false negatives. As casts to integral types are widespread (*e.g.,* a pointer passed as a `void*` argument), it would result in more instrumentation. Static analysis might reduce the number of classes to instrument by inferring if a `void` pointer is ever cast in its scope. Adoption would also require Sourcerer's performance overhead to be reduced through, for example, type check pruning [135] or type check removal [31] (§3.7). Code modernization, *e.g.,* removing casts on

hot paths or the source of instrumentation of classes heavily cached, has the highest improvement potential §3.6.3.

3.9 Conclusion

We introduce Sourcerer, a novel type confusion sanitizer that checks *all* casts at runtime. By combining inlined type information with our optimized RTTI initializer *RTTIInit*, Sourcerer checks all casts explicitly while reducing the divergence of the type++ dialect with the C++ standard. Additionally, RTTIInit supports unions and other idioms which were missing from existing tools.

We evaluate Sourcerer on the SPEC CPU benchmarks. Our tool checks twice as many unrelated casts compared to type++. On average, Sourcerer incurs 5.14% overhead, six times lower than the other complete type confusion sanitizer, EffectiveSan. Our ablation study identifies the cause of the overhead to be the required ABI changes. Lastly, during our fuzzing case study targeting specifically type confusions, we identify six new type confusion bugs and many code locations that would benefit from code modernization efforts.

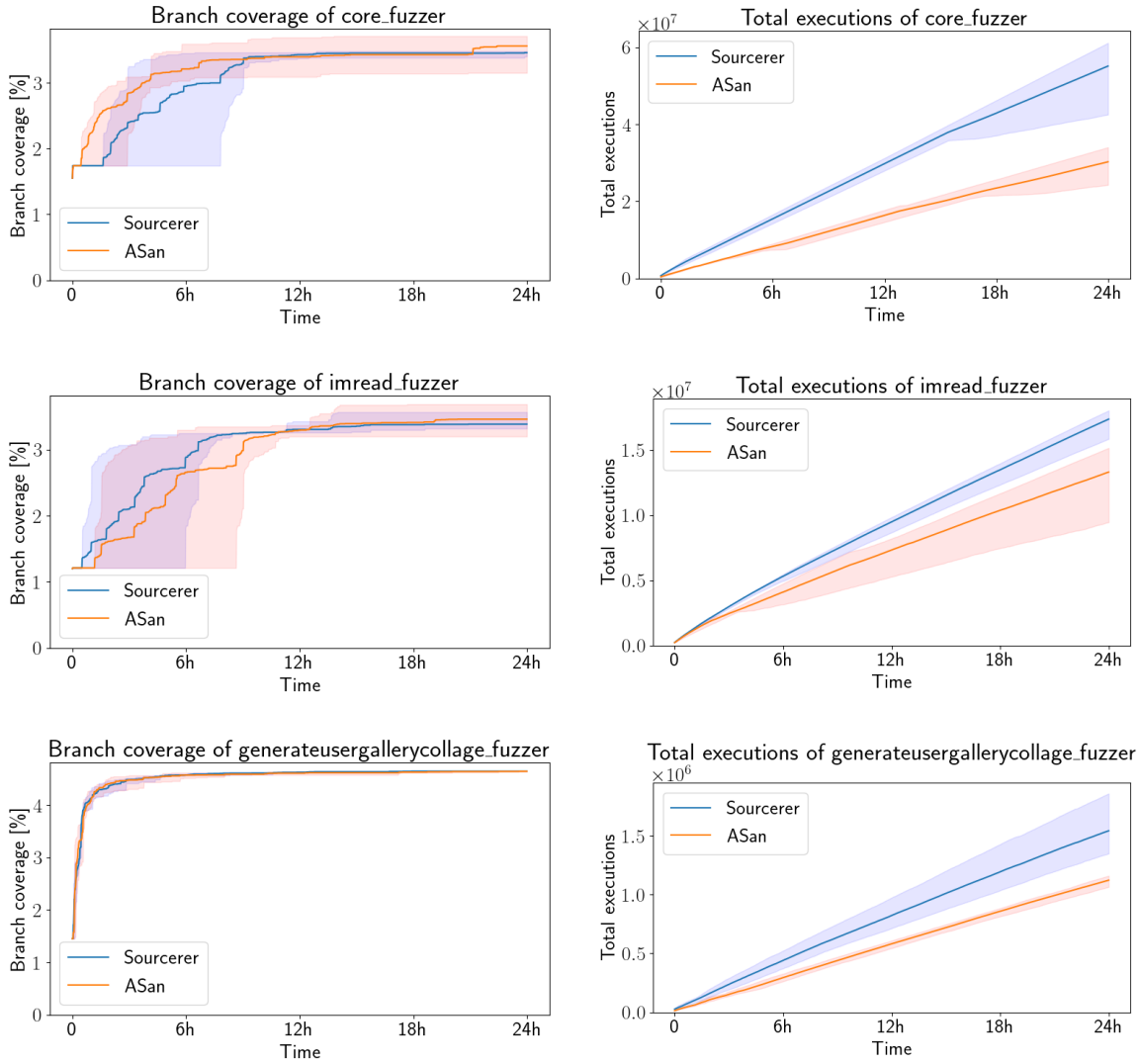

Figure 3.1 – Fuzzing campaign results. The left figures show the branch coverage throughout the 24h fuzzing campaign. On the right, Sourcerer allows for more executions due to reduced overhead compared to ASan.

Chapter 4

Liberating libraries through automated fuzz driver generation: Striking a Balance Without Consumer Code

(With a big smile) Gzzzz .... Grrrrii

Coline—4 month old fuzzer

Fuzzing has unparalleled bug-finding capabilities [33, 42, 133, 140]. This automated testing technique stochastically samples a program's input space, often thousands of times per second, to trigger flaws. General-purpose fuzzing engines expect a well-defined interface (*e.g.,* a `main` function) to run the code under test. This simple but effective design paved the way for a variety of generic and specialized fuzzers [16, 17, 19, 33, 42, 74, 93, 126, 128, 133]. Unfortunately, not all targets have fuzzing-compatible interfaces. In particular, user-space libraries (*e.g.,* `libpng` [101]) are designed to be integrated into stand-alone programs. Specifically, libraries expose a set of functions, the Application Programming Interface (API), to interact with the library code.

To tailor a fuzzer workflow to libraries, practitioners write small snippets of code (*i.e., drivers* or *fuzz drivers*) to tie the fuzzing engines to the library code. While some initiatives reduce the cost of fuzzing library drivers—*e.g.,* OSS-Fuzz [109], Google's effort to continuously test open-source libraries—driver creation remains, however, predominantly a manual effort [109]. Due to the required knowledge of the library's API and the scarcity of maintainers time, manually-written drivers are limited in the extent of their fuzzing campaigns as well as their capacity to evolve with the library changes. As we observe in OSS-Fuzz, fuzzed projects have generally reached a coverage plateau where the existing drivers no longer discover new functionalities. To overcome this limitation, academia and industry investigated approaches to create drivers automatically—trading maintainers' time for CPU resources—with the goal of covering new untested code paths and thereby exposing bugs in the targets [5, 18, 47, 56, 60, 132, 136, 137, 139].

The first explored approach to generate fuzz drivers is *consumer-dependent*, which analyzes the interaction of existing applications (*consumers*) with a library [5, 56, 60, 132, 136, 137, 139]. *Consumer-dependent* drivers are bounded to the patterns found in the consumers analyzed. To

overcome this limitation, *consumer-agnostic* techniques [18, 47] rely solely on the library code and apply static or dynamic techniques to infer valid library usages, thus finding bugs that may not appear in the known consumers. All current *consumer-agnostic* approaches, and some *consumer-dependent* works [56, 137], attempt to infer *valid library usage* ***and*** *valid inputs simultaneously*, thus solving two orthogonal problems at once, leading to a suboptimal solution due to the exponential growth of the input space. In practice, however, the computational budget is a finite resource T used to solve two distinct tasks: generate drivers (with the goal of maximizing potentially reachable coverage) and explore drivers through inputs (with the goal of maximizing concrete coverage and bug finding). In other terms, the time for driver generation (t_{gen}) and testing (t_{test}) is bounded by T, *i.e.,* $t_{\text{gen}} + t_{\text{test}} = T$. Manually written drivers allocate only testing time, while automatic driver generation mechanisms split testing and generation according to different policies. In Figure 4.1, we classify previous works according to their time budget allocation strategy.

LIBERATOR introduces a library testing model that balances resources between generation and fuzzing by leveraging three main techniques. First, *API sequence inference*: through static analysis, we infer which API sequences are more likely to contribute to coverage, allowing for fast and promising drivers. Second, through *dynamic pruning of ineffective sequences*, LIBERATOR avoids wasting time on unfruitful drivers. We propose to promptly discard broken API function sequences and avoid extending them further. Finally, since the aim is to test different code regions inside a library, we propose to *balance driver diversity to optimally distribute fuzzing energy* through a lightweight driver selection strategy that diversifies the API functions used.

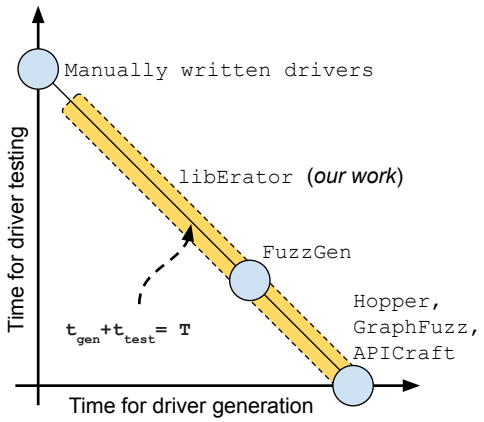

Figure 4.1 – The driver generation t_{gen} and the driver testing t_{test} sum up to the total computational time T, *e.g.,* $T = t_{\text{gen}} + t_{\text{test}}$. LIBERATOR (in yellow) can be configured with any balance of t_{gen} and t_{test} to fit the library.

We evaluate LIBERATOR by targeting 15 libraries representing varying API and input space complexity. As a baseline, we employ existing state-of-the-art solutions. We compare our approach against manually written drivers from the OSS-Fuzz project [109], *consumer-agnostic* works, HOPPER [18], and *consumer-dependent* works, namely UTOPIA [60], FuzzGen [56], and OSS-Fuzz-Gen [54]. Comparison with OSS-Fuzz demonstrates that LIBERATOR's drivers perform deep and complex library interactions, similar to the ones written by experts, *i.e.,* for six out of 12 libraries we achieve higher coverage. While, compared with HOPPER, we reach more coverage in eight out of the 13 libraries supported. Against consumer-dependent approaches, we achieve comparable coverage for UTOPIA despite their spurious use of internal API functions, and exceeds FuzzGen and OSS-Fuzz-Gen. Most importantly, we found 24 confirmed bugs in libraries already extensively tested by OSS-Fuzz, HOPPER, or OSS-Fuzz-Gen. We upstream fixes

for 21 of them and are in discussion with maintainers to contribute the respective drivers. Moreover, libErator experiences 25% of true positives, which doubles similar works [60]. These experiments demonstrate the capability of libErator to improve library testing capabilities.

In short, our key contributions are:

- A *consumer-agnostic* library model that balances driver generation and testing. Our model automatically generates valid library interactions and discovers code faults.
- We build our library model on three techniques: *API sequence inference* to proactively hint at promising sequences of API function calls, *dynamic pruning of ineffective sequences* to learn dysfunctional sequences, and *driver selection technique* to better distribute testing energy.
- libErator, an end-to-end framework that implements our *consumer-agnostic* library model and generates fuzz drivers for libraries by solely relying on their source code. We release the implementation of our prototype as open source, see Chapter 6.
- A detailed evaluation of our results against the state-of-the-art driver generation technique and manually written drivers, discussing the trade-offs of each approach.

4.1 Automatic Library Testing

Fuzz testing or fuzzing is a dynamic testing approach that leverages high execution throughput to sample the input space and discover bugs. Guided fuzzers [33, 133] leverage execution feedback to bias the input generation towards bug-prone code paths. These techniques have shown their effectiveness in industry [37, 41, 134] and are widely deployed, *e.g.,* thousands of projects are continuously fuzzed by OSS-Fuzz [109]. However, not all software is equally amenable to fuzzing.

Software libraries provide functionalities that, otherwise, should be re-implemented in each project with the risk of repeating past mistakes and bugs. Interactions between the main program and a library happens through functions part of the Application Programming Interface (API). To enable a fuzzer to test a library, an entry point, taking the form of a program, has to be created. These programs, called drivers, might require non-trivial inputs (*e.g.,* files) and should build the necessary state to interact, ideally, with the full library's API. libFuzzer [108], the pioneer in library fuzzing, models drivers as stub functions, called `LLVMFuzzerTestOneInput`, which takes as input a buffer of bytes. This flexible interface has been adopted as a standard in other popular fuzzing tools [33, 133]. Notably, libFuzzer executes drivers in a loop with different inputs to minimize the startup overhead. Therefore, drivers must exit cleanly, *e.g.,* by invoking `free` or closing files, to avoid lingering states leading to non-reproducible bugs. The number of drivers remains limited as they are manually written, *e.g.,* only *eight* out of the 15 libraries evaluated have multiple drivers.

```
1  extern "C" int LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
2   vpx_codec_ctx_t codec;
3   vpx_codec_dec_cfg_t cfg;
4   size_t frame_size;                                                          ①
5   uint8_t *frame; /* size(frame) == frame_size */
6    ...
7   cfg = ... /* from data */
8   frame_size = ... /* from data */
9   frame = malloc(frame_size);                                                 ②
10  memcpy(frame, /* from data */, frame_size);
11  /* codec cannot be populated from data! */
12   ...
13  vpx_codec_dec_init(&codec, VPXD_INTERFACE(DECODER), &cfg, 0);               ③
14  const vpx_codec_err_t err = vpx_codec_decode( &codec,frame, frame_size, nullptr, 0);
15   ...
16  vpx_codec_destroy(&codec);                                                  ④
17  return 0;
18 }
```

Listing 4.1 – Simplified example of a C driver from `libvpx` highlighting the four driver regions: ① variable definition, ② input transfer, ③ API chain execution, and ④ state cleanup.

Listing 4.1 presents an abbreviated driver written by the `libvpx` maintainers [46]. Without loss of generality, we decompose the driver in *four* regions: ① variables definition, ② input transfer, ③ API chain execution, and ④ cleanup. Region ① declares the necessary variables for the driver. Region ② bridges the fuzzing input into the respective variables. However, not all the variables can be populated with raw data, *e.g.,* `codec` has internal pointers fields and requires invocation of `vpx_codec_dec_init`. Region ③ consists of the *API chain*, *i.e.,* a valid sequence of API calls. *API chains* are composed of a valid sequence of function calls and correct variables as their respective arguments. Finally, region ④ cleans the driver state to avoid memory leaks, *e.g.,* `vpx_codec_destroy` releases `codec` at Line 16.

4.2 Challenges for Automatic Driver Generation

Driver generators aim at inferring valid and interesting API Chains (§4.1). However, not all API function call combinations are valid. Producing *valid* chains introduces challenges exemplified through a combinatory exercise. Given a library that exposes N API functions, an under-approximation of the number of possible API Chain is given by $N^{|API\_Chain|}$. This means that the search space grows exponentially with the chain size. This approximation is conservative as it does not consider function arguments. On top of that, each API Chain has its own input space that can be modeled as a bitstream of length I. On classic targets, the fuzzers only sample the

input space (*i.e.,* 2^I). Conversely, in automatic library testing, the system is also responsible for generating drivers, thus extending the search space to at least $N^{len(API\_Chain)} \times 2^I$. Therefore, creating valid API Chains is an additional dimension requiring ad-hoc reasoning. We study how current approaches model library testing, and propose a new solution to better address this extra dimension.

Current works generate drivers by synthesizing API Chains *and* searching for valid inputs simultaneously [18, 47, 56, 137]. They assume coverage to be a fair feedback, *i.e.,* API Chains reaching more coverage should be expanded, leading to critical limitations. First, API Chains may face a coverage wall, meaning they do not reach new coverage, however, the fuzzer needs more time to pierce through the perceived coverage wall. When this occurs, current works misjudge the API Chain as unpromising and stop testing it. Second, libraries may require a long sequence of API function calls to initialize complex structures to unlock access to deep library code. However, prefix API Chains may not reach meaningful coverage and thus be ignored, despite being crucial for longer API Chains. Third, current works produce new API Chains continuously, without testing them deeply. Hence, an explosion of undertested API Chains hinders the overall testing progress.

By surveying the state-of-the-art, we define three challenges that drive LIBERATOR design.

(C1) Predicting complex API combinations. To reduce the computational cost associated with the generation, it is crucial to predict which API Chains are interesting.

(C2) Avoiding coverage wall biases. Come up with a strategy to learn, and avoid building on, unfruitful API Chains that should not suffer from biases created by coverage walls.

(C3) Deep testing of API Chains. To efficiently fuzz, LIBERATOR needs to avoid redundant testing of similar API Chains and invest computation cycles on diverse library usages.

4.3 Driver Specification

In this section, we detail the characteristics of the generated drivers. LIBERATOR's prototype creates functional and valid drivers written in C, compatible with existing fuzzers like libFuzzer [108] or AFL++ [33]. Therefore, LIBERATOR needs to use coherently the variables passed to the API functions. This constrains some technical choices, such as the type system and the handling of the variables lifetime. Most of our solutions extends ideas from previous works [18, 56]. LIBERATOR's design is language-agnostic and adaptable to other scenarios.

Type System. LIBERATOR's type system leverages previous works insights [18, 56], which are summarized in Table 4.1. Specifically, LIBERATOR triv-

Table 4.1 – LIBERATOR's partition of the type system. The "Source" column indicates the source of information necessary for the analysis, while the other columns describe which properties they possess: whether they can be referenced (R), allocated (A), or initialized from fuzzing input (I).

Type	R	A	I	Source
Primitive Types	✓	✓	✓	Header Files
Primitive Arrays	✓	✓	✓	Header Files
Strings	✓	✓	✓	Header Files
File Paths	✓	✓	✓	Library Code

ially handles primitive types (*e.g.,* `int`, `char`). For pointers, we try to infer the length of the underlying array through a data-flow analysis. For dynamically sized arrays, the driver allocates a buffer (*i.e.,* through `malloc`), we extend this approach to multidimensional arrays. We verify if some function argument of specific types (*e.g.,* `uint`, `size_t`) control the size of dynamic allocations, and bound their value to avoid out-of-memory errors [123]. In the case of `char*`-like types, we terminate the buffer with `NULL`. Additionally, through data-flow analysis, we infer if chars arrays are used as file path in known system functions (*e.g.,* `fread`), and, in such case, allocate a temporary file to store the fuzz input. Moreover, libErator handles *complete* and *incomplete* structures. *Incomplete* types lack information regarding their size [96], and therefore cannot be allocated. Consumers can only have pointers to *incomplete* types and use the library's APIs to handle their lifetime. *E.g.*, at Line 13 in Listing 4.1, the second argument of the function `vpx_codec_dec_init` is incomplete and can only be created through the `VPXD_INTERFACE` function. *Complete* types, instead, may need non-trivial API Chains to be properly initialized, as in the case for `vpx_codec_ctx_t`, which needs a correct sequence of API calls (Line 13). Lastly, we handle function pointers by synthesizing empty stub *callback functions* from the API source code, and using their address in the API function calls.

Lifetime Properties. Drivers need to handle their internal variable lifetime. Specifically, we support the three variable lifetimes of C: *local*, *dynamic*, and *static*. *Local* variables are allocated in the driver's stack frame and released once the driver terminates its execution. *Dynamic* variables must be allocated and freed coherently. To this end, our analysis mark library functions that *create* variables, *i.e.,* returning allocated structures (*e.g.,* via `malloc`), and functions that *delete* variables, *i.e.,* function arguments passed to `free`. If an API function returns a pointer to a global reference (*e.g.,* BSS), we consider the memory location as *static*. API functions that *create*, *delete*, or return *static* reference hint at inter-API dependency. *E.g.*, an API function that allocates objects should appear early in the driver. Likewise, API functions that `free` objects should appear during cleanup.

Var-Len. libErator supports API arguments dependency between a variable and its length (*Var-len*). Two API arguments, called V and L, are in a *Var-len* relation if V points to buffer and L indicates V's size, *e.g.,* in Listing 4.1, the function `vpx_codec_decode` requires `data` to be a buffer of bytes (V) of size `frame_size` (L). We ensure that *Var-len* arguments are used coherently. Previous works already investigated *Var-len* properties [18, 60], and we expand on their insights.

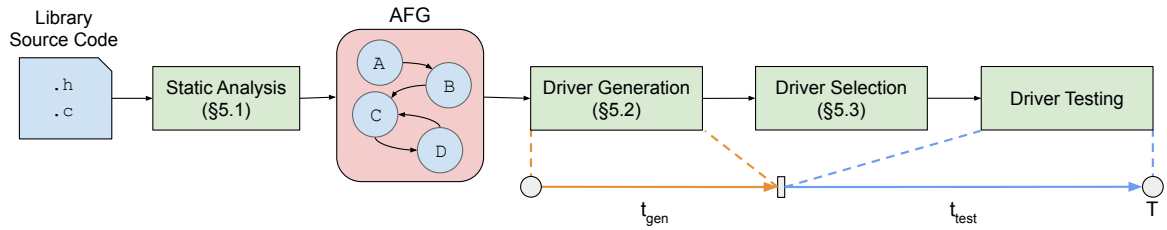

Figure 4.2 – LIBERATOR's architecture. The library source code is analyzed to model the API function arguments and the type system into the API Flow Graph (AFG). Then, the driver generation module synthesizes new drivers and select a diverse set. Finally, the drivers are tested in a fuzzing campaign.

Loop-Index: When a `for` or `while` loop traverses a buffer V, the loop index L represents its upper bound (*e.g.,* `while(i<L) V[i]`). To this end, we develop an inter-procedural analysis, while UTOPIA employs an intra-procedural approach.

Buffer Directly Controlled: Libraries may use the argument L to calculate the last valid address of the buffer V (*e.g.,* `last_V=&V[L]`) and iterate until then. LIBERATOR is the first to support these cases.

Third-party Functions: External functions may manipulate buffers, *e.g.,* `memcpy`. If V and L are used as source and size in `memcpy` (*e.g.,* `memcpy(V,data,L)`), then we assume a *Var-len* relation. Our analysis first locates the buffer arguments used in transfer data functions and then checks if any other API function argument is used as a size parameter. Our prototype handles the main `libc` functions known to transfer/alter buffers, *e.g.,* `strcpy`, `memcpy`, and `memset`, and allows for the integration of third-party functions.

4.4 libErator Design

LIBERATOR's end goal is to produce fuzz drivers that (i) can be used in production, and (ii) exhibit diverse library interactions. LIBERATOR's design centers around the challenges introduced in §4.2. Each challenge is addressed by a dedicated technique implemented as a module, see Figure 4.2. All the modules refer to a central data structure, called *API Flow Graph* (AFG), that encodes API function information. Specifically, the AFG provides hints for creating long API chains thus addressing challenge C1. The AFG is produced by the static analysis module (§4.4.1) taking only the library source code as input. By leveraging the AFG, a driver generator module (§4.4.2) produces new drivers through an algorithm that learns to avoid invalid API Chains, *i.e.,* API function sequences that do not adhere to the library semantic. This improves the rate of correct drivers without suffering from coverage wall biases, thus tackling the challenge C2. Then, LIBERATOR uses a lightweight clustering algorithm to select drivers that stress different library

regions (§4.4.3), thus enabling longer fuzz driver testing and satisfying challenge C3. Finally, all selected drivers are fuzzed for an identical period with an initial corpus bootstrapped from the seeds produced during the generation phase.

libErator's configuration only requires the exported library's headers and a build script for the library. The generated fuzz drivers follow the specification in §4.3.

4.4.1 Static Analysis

The purpose of the static analysis is to populate the *API Flow Graph* (AFG), the directed graph encoding the API function dependencies and helps predict interesting API Chains. We use a static data-flow analysis to infer non-trivial properties from the library code. When re-using existing techniques, we point to the implementation (§4.5) and otherwise describe our analysis in details.

Algorithm 1: Driver Creation

Input: The AFG: D, Time for generating drivers: t_{gen}
Output: A list of drivers to be deeply tested

```
1 drivers_history ← {};
2 while time_spent() < t_gen do
3 |   driver ← generate_api_chain(D,
  |     drivers_history);
4 |   status ← probe(driver);
5 |   drivers_history ←_add (driver, status);
6 end while
7 return extract_positive_drivers(drivers_history)
```

API Flow Graph (AFG). The graph is built from the function signatures, exposed in the library header files. The AFG represents an over-approximation of the possible *API Chains* [18, 56, 137], and guides the driver generation (§4.4.2). Specifically, the graph's nodes represent an API function, and their outgoing edges are possible subsequent API function calls. For each API function A, we define its inputs and outputs, called $I(A)$ and $O(A)$, respectively. Inputs $I(A)$ are the arguments passed to the function, while output $O(A)$ are the return values and the arguments passed by reference. Then, we define B *depends_on* A if the types of $O(A)$ and the types of $I(B)$ have a non-empty intersection, *i.e.,* B *depends_on* $A \iff O(A) \cap I(B) \neq \emptyset$. Based on the *depends_on* relation, we connect node B to node A in the AFG. By traversing the AFG, libErator chooses the next function to append to the API Chain. Then, through static and dynamic analysis, the driver generator module prunes invalid API Chains. In the following, we detail the additional information contained in the AFG.

AFG Bias. libErator uses information obtained from the static analysis to guide (bias) the API Chain creation (§4.4.2). We observe that libraries commonly use simple functions as preliminary steps for more complex library interactions later on. For instance, `LibHTP`'s drivers require at least three API functions to set up a valid `htp_connp_t` structure necessary for interaction with more complex library functionalities. Unfortunately, the API functions used to build up `htp_connp_t`

reach only a shallow coverage. Therefore, state-of-the-art driver generation approaches leveraging code coverage as feedback may not prioritize these functions. Conversely, our static analysis infers signal from the source code as functions setting up more complex library usage manipulate the fields of the desired structure. In particular, we enumerate all the fields and subfields that are set or retrieved, carefully handling recursive structures. Primitive types are assimilated to structures with a single field. The total number of manipulated fields is used to bias the driver generation module when traversing the AFG. By observing how a function manipulates a structure, libErator can foresee more complex API Chains.

4.4.2 Driver Generation

The driver generation module relies on the AFG to generate functional drivers (§4.1). More specifically, the module uses a dynamic generation strategy, in which, increasingly long API Chains are generated and tested. algorithm 1 describes the overall strategy. Crucially, each new API Chain is probed in a short fuzzing campaign (*i.e., five* minutes). The probing reveals if the driver interacts with the library, *i.e.,* it produces *seeds*. Drivers producing seeds are marked as *positive*, while the others are marked as *negative*. All the API Chains are recorded in a *driver history* that is used in combination with the AFG for generation. This mechanism overcomes the limitation of previous works that use coverage to promote drivers. Since drivers often face a coverage wall, short fuzzing campaigns cannot faithfully assess drivers' quality. Conversely, malformed drivers hardly show any interaction (seed), giving more reliable feedback. Furthermore, negative chains allow the system to preemptively avoid AFG paths leading to useless drivers. This algorithm "learns" valid API Chains and discards unfruitful ones, thus, addressing challenge C2. Crucially, choosing seeds over coverage also boosts performance, as computing coverage requires resource intensive tools like SanCov [119].

Our algorithm generates an API Chain by traversing the AFG. Throughout the traversal, the algorithm respects the library control- and data-flow constraints encoded in the AFG. During this phase, we use the static analysis results and the *driver history* to bias the AFG traversal towards interesting chains, while avoiding repeating malformed drivers. Then, we produce the code for both the input transfer and cleanup regions of the driver.

API Chain Creation. algorithm 2 describes the process, which takes the AFG and the *driver history* as inputs. In the first part, libErator identify *source* functions (Line 1–5). We define an API function as *source* if all its arguments can be readily allocated and initialized. *Source* functions represent the AFG entry points. Consequently, we traverse the AFG and try to append new function calls to the chain (Line 7–16). Additionally, we use the *driver history* to discard chains already evaluated as malformed. The algorithm retrieves from the AFG the number of manipulated fields to bias its selection toward instantiable API functions. If no candidate API

Algorithm 2: API chain creation

```
   Input: The AFG: D, and drivers_history
   Output: The API chain and the list of variables
1  source_api ← get_source_api(D);
2  good_api_chains, bad_api_chains ← split_positive_negative_chains(drivers_history);
3  api ← random_bias(source_api, D);
4  api_chain = [api];
5  var_alive ← try_to_instantiate(api, ∅);
6  while true do
7      candidate_next_api_fun = []; /* Search API functions to add to the chain      */
8      foreach n_api ∈ D[api].adjacency_list do
9          if api_chain ∪ [n_api] ∈ bad_api_chains then
10             continue; /* Avoid repeating known failed drivers                      */
11         end if
12         new_var ← try_to_instantiate(n_api, var_alive);
13         if is_valid(new_var) then
14             candidate_next_api_fun ←_add (n_api, new_var);
15         end if
16     end foreach
17     if len(candidate_next_api_fun) == 0 then
18         api ← random_bias(source_api, D); /* If no valid candidates, pick a new source */
19         var_alive ← try_to_instantiate(api, var_alive);
20     else
21         (api, var_alive) ← random_bias(candidate_next_api_fun, D);
22     end if
23     api_chain ← api_chain + [api]; /* Update the chain, if new, stop and probe it   */
24     if api_chain ∉ good_api_chains then
25         break;
26     end if
27 end while
28 return (api_chain, var_alive)
```

function is available, it selects a new *source* function (Line 17–22). The algorithm terminates when it creates a new chain that has never been probed, *i.e.,* it is neither *positive* nor *negative* (Line 24). The idea is to start by exploring the *source* functions and then expand while avoiding chains known to be malformed.

The AFG traversal is controlled by an instantiation routine (*i.e.,* `try_to_instantiate`), which is an oracle that answers the question: *Can we invoke the given API function given the current variables (var_alive)?* The instantiation routine decides which API functions can be appended to a given *API Chain*. Formally speaking, the routine has *three* duties: (1) try to reuse already instantiated variables, (2) generate new variables if their type allows it, (3) update `var_alive` to track the variable lifecycle (*e.g.,* for the final cleanup). More importantly, the instantiation routine traces the variable lifetimes and avoids reusing the deallocated ones. Additionally, it coherently associates variables used in *Var-len* arguments and uses additional variables to handle variadic arguments [23].

4.4.3 Driver Selection

The driver generation module (§4.4.2) produces a list of *positive* drivers (Line 7). These harnesses are sufficiently stable to be tested, however, can be numerous with up to 100 drivers generated per hour depending on the library. Deep testing each of them is, therefore, infeasible. This is even more problematic when the time budget is constrained, for example during a fuzzer evaluation.

Therefore, the strategy to select the most relevant drivers should satisfy the following requirements. First, it needs to be lightweight. Second, it should select drivers diversifying library usage. While coverage distinguishes drivers that reach different code regions, the short fuzzing period is insufficient to overcome coverage walls, making coverage an unreliable metric. Moreover, calculating fine-grained coverage is resource intensive, stealing resources from the actual testing.

To address this problem, we devise an algorithm around the intuition that API functions are a proxy for library regions. Specifically, we employ a lightweight clustering algorithm based on affinity propagation algorithm [36] and Levenshtein distance [131] to automatically group drivers based on the API functions they use. We treat the API Chains as a list of symbols, where each symbol is an API function, and calculate the Levenshtein distance between each pair of chains. Then, the affinity propagation automatically clusters chains with closer distance. For each cluster, we select the drivers that produced the most seeds, as we deem them more promising for longer testing. Despite affinity propagation having $\mathcal{O}(n^2)$ complexity, we process thousands of API Chains in less than a minute, thus fitting our leanness requirement. As a result, our driver selection algorithm finds relevant targets efficiently, maximizing the resources devoted to testing the library and, thus, addressing the challenge C3.

4.5 Implementation

Prototype implementation. libErator analysis leverages SVF [115, 130], complemented with 4K LoC of C++ to extract the dependency graph. The rest of the tool is composed of around 5K LoC of Python code. libErator generates drivers as C programs, subsequently compiled and statically linked with the target library. The drivers' input follow the format from libFuzzer. libErator, however, can be used with other fuzzing engines, *e.g.,* AFL++.

Static analysis. The static analysis used in §4.3 and §4.4.1 is based on the default Use-Def graph provided by SVF [115, 130] using SVF's Flow-Sensitive point-to analysis [51]. However, we observe that the SVF analyses do not correctly resolve indirect calls for global function pointers. Usually, global function pointers are used to set at runtime system-dependent functions (*e.g.,* `malloc`/`free`) through specific environment variables. This limitation leads to an incomplete call

graph that hides important code patterns. To cope with this issue, we locate (a) all the global structures containing function pointers, and (b) all the code locations where the global function pointers were set. Finally, we use this information to infer missing target sets and update the call graph accordingly. This simple strategy is sufficient to handle the analyzed libraries.

Type Length Value (TLV) implementation. The drivers synthesized by LIBERATOR assume that some variables have a fixed size (*e.g.,* `char s[10]`) while others have a dynamic size (*e.g., Var-len*). On the other hand, the fuzzing engine produces "random" inputs. Therefore, the fuzzer engine and driver must agree on an input structure to correctly mutate and bridge it into variables. To solve this problem, we encode inputs as a TLV structure. More precisely, the driver contains a custom mutator—`LLVMFuzzerCustomMutator` routine in our prototype [44]—that mutates the input according to the TLV encoded structure and maps it to the driver's variables.

4.6 Evaluation

We evaluate LIBERATOR by answering the following research questions:

RQ1: How does the trade-off between t_{gen} and t_{test} manifest itself in practice (§4.6.1)?

RQ2: To which extent does LIBERATOR explore libraries? (§4.6.2)?

RQ3: How does LIBERATOR compare to state-of-the-art library fuzzing tools (§4.6.3)?

RQ4: Can LIBERATOR find bugs in real-world libraries (§4.6.4)?

RQ5: How does each component of LIBERATOR contribute to its performance (§4.6.5)?

Compared Works. Our evaluation compares LIBERATOR against state-of-the-art *consumer-agnostic*—HOPPER [18]—and *consumer-dependent* tools—UTOPIA [60], FuzzGen [56], and the Google framework OSS-Fuzz-Gen [53, 54]. We select these works based on the availability of either their artifact or their released drivers. Additionally, we select all manually written drivers from the OSS-Fuzz project [109] and the projects' repositories to provide a comparison with existing drivers.

Benchmarks Selected. We evaluate LIBERATOR on 15 libraries ranging from 8K to 518K LoC, as listed in Table 4.2. We choose all C targets from HOPPER and UTOPIA apart from five libraries which SVF does not support (§4.7). Additionally, we include the four libraries from OSS-Fuzz-Gen with the most drivers. All the libraries are tested on their most recent commits except when

Table 4.2 – Targets selected for LIBERATOR evaluation. "#API Func." denotes the number of exposed API functions in the library. The last column reports the duration of LIBERATOR static analysis.

Name	K LoC	#API Func.	Duration
c-ares	55.99	61	1 min 1 s
cJSON	16.57	78	24 s
cpu_features	8.36	7	43 s
libaom	518.38	47	2 h 40 min 39 s
libdwarf	126.83	333	2 h 51 min 26 s
LibHTP	38.59	249	13 min 43 s
libpcap	45.59	89	42 s
libplist	11.25	101	47 s
libsndfile	56.42	40	38 min 43 s
LibTIFF	87.16	196	7 h 32 min 5 s
libucl	17.04	125	2 h 38 min 25 s
libvpx	362.05	38	2 h 19 min 43 s
minijail	18.87	95	20 s
pthreadpool	12.69	30	1 min 27 s
zlib	29.94	88	21 s

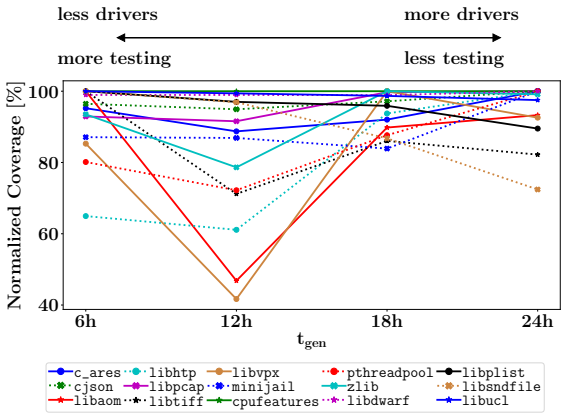

Figure 4.3 – Normalized coverage in five-runs average achieved by LIBERATOR drivers as a function of t_{gen}. The different behaviors of the libraries as t_{gen} grows, highlight the need for a tool where t_{gen} and t_{test} can be fine-tuned. The raw numbers are in Table 4.4.

comparing with UTOPIA where we use the versions from their evaluation since the artifact is otherwise incompatible.

Experimental Setup. LIBERATOR evaluation was performed in Docker containers based on Ubuntu 20.04 on an Intel Xeon Gold 5218 @ 2.30 GHz CPU with 64 GB of RAM. Every library and driver is compiled with identical options. Each result is the average of *five* experiment repetitions.

Fuzzing Setup. For all fuzzing campaigns, we instrument libraries and drivers with ASan [106] and SanCov [119]. To measure the coverage, we replay the corpus and count only the branches and lines traced by SanCov in the library while discarding the coverage in the driver itself. Finally, all the testing campaigns are launched with an empty initial corpus.

4.6.1 RQ1 - t_{gen} vs t_{test} Trade-off Analysis

Automated library fuzzing faces an inherent trade-off between the computation time invested in generating new drivers and the time spent testing them. To demonstrate and quantify this trade-off, we evaluate the overall performance—in terms of code coverage—of fuzzing campaigns with different balances of t_{gen} and t_{test}. In particular, we perform *four* 24-hour campaigns for t_{gen}

values of 6-, 12-, 18-, and 24-hours. As the time necessary for selection is minimal, t_{test} is the complement to 24 hours of t_{gen}. If t_{test} is *zero*, we measure the cumulative SanCov coverage produced by the drivers during the driver generation. Otherwise, the coverage is measured during the fuzzing campaign lasting t_{test} hours.

The results of these campaigns are presented in Figure 4.3. Our main observation is of distinct behaviors among the tested libraries. While `minijail` and `cJSON` reach more coverage while increasing t_{gen}, `libaom` and `LibTIFF` show better results for the smallest t_{gen}, *i.e.,* 6 hours. This correlates with the complexity of inputs accepted by the libraries. For instance, the core of both `LibTIFF` and `libaom` is to parse complex media formats, which the fuzzer is unlikely to synthesize correctly initially. With longer testing time, the fuzzer learns the input format and provide better inputs to the drivers. `cJSON` and `minijail` inputs have simpler structures, and testing benefits more from a diverse set of drivers which allows for broader coverage. Looking at `libpcap` and `zlib`, the coverage plateaus or decreases beyond a certain t_{gen}. Likely explanations are either a coverage wall or the saturation of the driver corpus—*i.e.,* the campaign does not benefit from new drivers anymore. For `libaom` and `libvpx`, we observe a sharp drop in coverage for a t_{gen} of 12 hours. We hypothesize that this setting exemplifies the worst of both worlds: the fuzzer does not have enough time to learn the input structure, and we lack enough driver diversity to trigger more code paths.

In conclusion, Figure 4.3 shows the absence of a one-size-fits-all for automated library fuzzing. The optimal balance between t_{gen} and t_{test} depends partially on the API complexity and its input structure. This motivates the need for a tool configurable to any balance between t_{gen} and t_{test}. libErator is designed around controlling this trade-off and can, therefore, be used to find the best resource allocation for fuzzing a specific library.

4.6.2 RQ2 - How does libErator Test Libraries?

We investigate how libErator performs library testing and highlight different aspects of the *valid* drivers synthesized during the generation process (§4.4.2). To this end, we measure the cumulative coverage and the API functions invoked during a campaign where t_{gen} is set to 24 hours. Specifically, the cumulative coverage is calculated by merging the progressive SanCov profiles [118]. Both metrics are expressed in terms of number of drivers generated. If a repetition has fewer drivers, we take its total cumulative coverage for the outstanding driver average.

Figure 4.4 shows the cumulative coverage after 24 hours. Similarly to standard fuzzing, the coverage tends, for most libraries, to reach a plateau. This signifies that generating additional drivers would, likely, bring only marginal new coverage. This plateau can either highlight the need to switch to more prolonged fuzzing or a coverage wall. Therefore, switching to prolonged fuzzing may not always be beneficial, as will show §4.6.3. Libraries that expect simple inputs

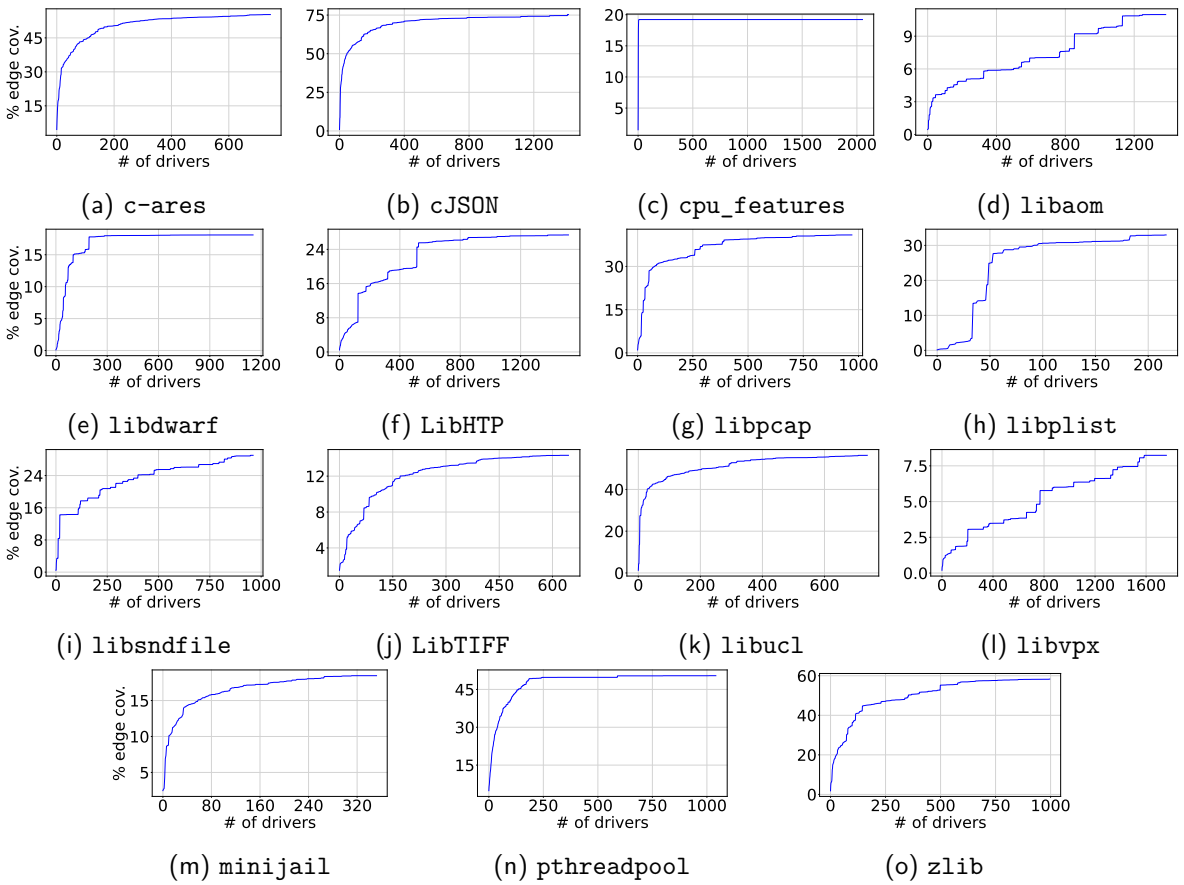

Figure 4.4 – Cumulative edge coverage reached as drivers are generated over 24 hours. The X-axis is the number of drivers generated during the 24 hours fuzzing session. The Y-axis is the cumulated edge coverage reached.

(*e.g.,* integers or strings) benefit more from shorter campaigns with different drivers, as sampling interesting inputs for these types is simpler than for complex structures.

Figure 4.5 shows the percentage of API functions tested during the driver generation. As expected, the libraries showing a clear plateau in terms of coverage (Figure 4.4), *e.g.,* `cJSON`, also tend to have exhausted the API functions. Conversely, `minijail`, `LibTIFF`, and `LibHTP` did not reach a clear plateau and LIBERATOR might explore new API Chains combinations given more time. Notably, libraries requiring complex inputs and complex API Chains, such as `LibTIFF` and `libaom`, benefit from both a longer driver generation and longer testing.

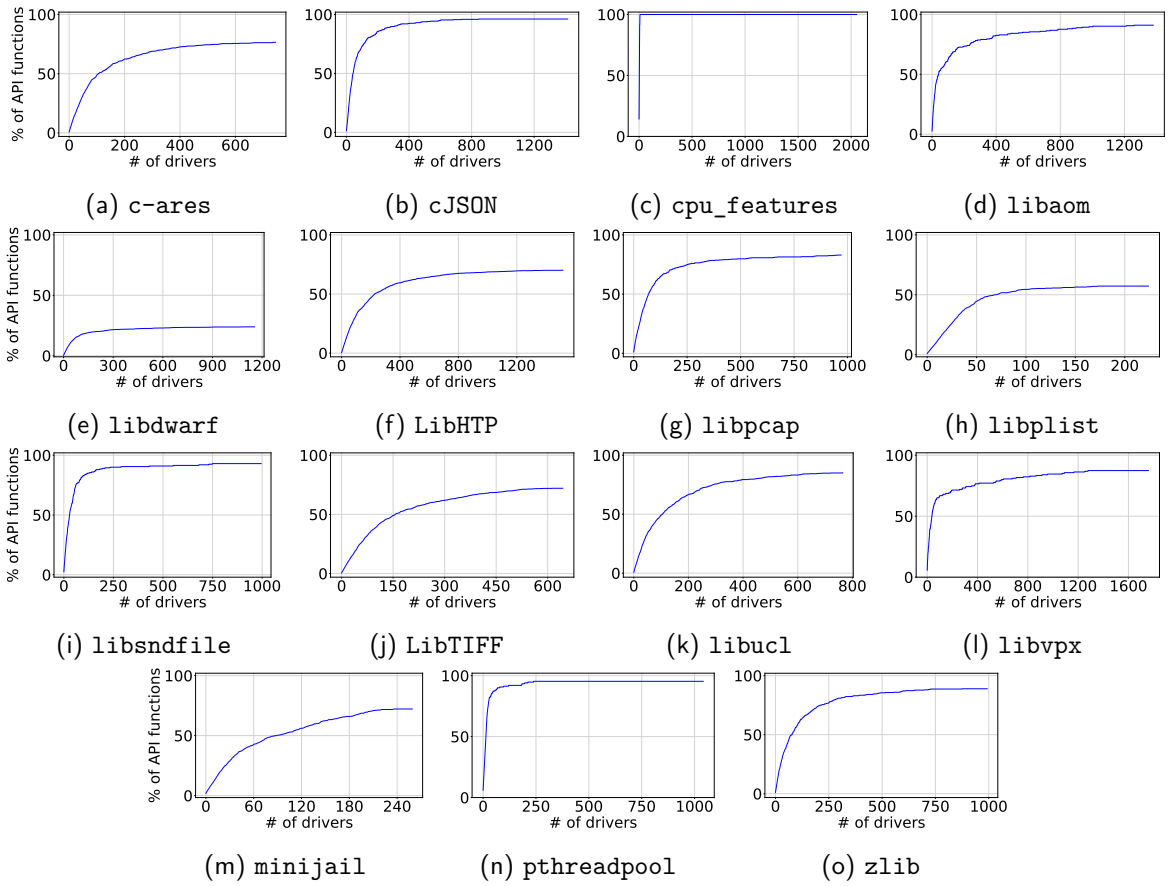

Figure 4.5 – API function coverage over 24 hours driver generation. The X-axis is the number of drivers selected during the generation. The Y-axis is the percentage of API functions covered as more drivers are created.

4.6.3 RQ3 - Comparison with State-of-the-art

To assess LIBERATOR, we compare against publicly released drivers and functional artifacts from state-of-the-art library fuzzing tools. The five works selected have explored library fuzzing from different perspectives and with varying assumptions. Table 4.3 summarizes their characteristics and highlights their differences. Most tools share characteristics, *i.e.,* they operate at source code level, are *consumer-dependent*, and use existing sanitizer (*i.e.,* ASan [106]). Only UTOPIA and HOPPER stand out. UTOPIA transforms libraries' unit tests into fuzz drivers. However, as unit tests often interact directly with internal library functions, it biases the amount of reachable code, as observed in LibHTP. HOPPER works with stripped binaries, thus prohibiting the use of ASan, and, instead, employs binary re-writing to implement sanitization. Their approach suffers, however, from false negatives (*e.g.,* it misses out-of-bound memory accesses). This leads HOPPER to measure coverage past runtime errors in code unreachable with ASan thereby artificially inflating coverage numbers.

Table 4.3 – libErator's features compared with state-of-the-art works. UTopia and Hopper are the two main outsiders. UTopia does not adhere to the fuzz driver definition and operates on library functions outside of the public API. Hopper works at the binary level and does not use ASan but an incomplete custom sanitizer.

Tool	Consumer Relation	Standard Sanitizer	Use API Only
UTopia	Dependent	✓	
OSS-Fuzz	Dependent	✓	✓
OSS-Fuzz-Gen	Dependent	✓	✓
FuzzGen	Dependent	✓	✓
Hopper	Agnostic		✓
libErator	Agnostic	✓	✓

To measure this impact, we replay Hopper's queue with ASan and observe that 73.39% and 8.05% of the inputs for `libplist` and `libdwarf` respectively, suffer from runtime errors. From our analysis, we conclude that Hopper and libErator operate on different assumptions (open versus closed source) resulting in coverage measurements not directly comparable.

Table 4.4 shows the results of libErator across different t_{gen} and t_{test} as well as the results of Hopper and OSS-Fuzz. Table 4.5 provides a comparison against UTopia, while and Table 4.6 compares libErator to FuzzGen and OSS-Fuzz-Gen. The following paragraphs describe each comparison in detail. The coverage reported is only the library coverage after a time budget of 24 hours. We select the best libErator configuration for each library and compare its coverage with the other tools.

Hopper. Despite the difference in sanitization, libErator outperforms Hopper on eight out of 13 libraries. We carefully examine the five libraries where Hopper prevails. In general, Hopper benefits from its faster generation of drivers. This affects the exploration of `LibHTP`, which is the largest library tested in terms of numbers of API functions in our evaluation set (see Figure 4.2). By generating vastly more API Chains than libErator, Hopper can reach shallow coverage in many more functions resulting in higher coverage overall. This also explains the reached coverage in `c-ares` and `libucl`. For `cJSON`, we observe that Hopper avoids a recurrent false positive use-after-free error that hinders libErator from reaching a higher coverage. For `libplist`, Hopper's sanitizer does not detect a runtime error, thus not stopping library testing. Indeed, we observe that 73.39% of the inputs generated by Hopper for `libplist` would be stopped in libErator by ASan.

OSS-Fuzz. Compared to OSS-Fuzz, libErator covers more code on six of the 12 compatible libraries. For complex APIs like `libaom` and `libvpx`, drivers need complex code structures such as loops and conditions for probing deeper code paths. This limitation is common to all

other automatic approaches, *i.e.,* HOPPER, OSS-Fuzz-Gen, FuzzGen, and UTOPIA. OSS-Fuzz's drawback is that the current drivers are exhaustively tested and without extensive manual efforts can hardly reveal new bugs, as we study in §4.6.4.

UTopia. Table 4.5 shows the comparison with UTOPIA. As UTOPIA converts existing unit tests into harnesses, its drivers interact through library functions absent from the public API, thus giving a substantial advantage in terms of coverage. This is particularly evident for `minijail` and `LibHTP`. For `libaom`, UTOPIA's drivers are more stable since they include loops and more complex code structures. Nonetheless, LIBERATOR overcomes UTOPIA in half of the targets without requiring any unit tests and respecting the intended library interaction. Moreover, our evaluation of UTOPIA results only in false positive crashes which are cumbersome to triage.

OSS-Fuzz-Gen. Table 4.6 reports the comparison with OSS-Fuzz-Gen. Since we lack the resources to run the Large Language Model (LLM), we test the publicly released drivers [54]. Overall, LIBERATOR outperfoms OSS-Fuzz-Gen on five out of six libraries while coming close in the remaining one. OSS-Fuzz-Gen is penalized by the small number of API functions included in its drivers. Indeed, the prompts used insist on keeping the driver concise to avoid compilation errors. Conversely, LIBERATOR benefits from interaction with diverse API functions thanks to §4.4.3, thus exercising more diverse parts of the library's code.

FuzzGen. Table 4.6 presents the results of FuzzGen. Due to compilation errors in the generation pipeline, we fall back to the published drivers overlapping with our target set, namely `libaom` and `libvpx`, note that both drivers required manual patches to compile. Despite our investigation, we could not identify a clear cause for the low coverage in `libaom`. For `libvpx`, our patch corrected an initialization issue for the `vpx_codec_ctx` structure, which is essential for library interaction. Resolving this issue allowed FuzzGen to achieve a coverage comparable to LIBERATOR.

Overall, LIBERATOR outperforms HOPPER on most libraries and on almost all libraries compared to OSS-Fuzz-Gen and FuzzGen. Compared to UTOPIA, LIBERATOR reports, on average, a comparable coverage despite using only publicly available API functions. Finally, LIBERATOR complements OSS-Fuzz by providing a broader and evolving exploration of the libraries. We therefore conclude that automatically creating high-quality drivers without consumers or test cases is achievable through analysis of the library's source code alone.

4.6.4 RQ4 - libErator Bugs Finding Capabilities

We compare the bug-funding capability of LIBERATOR against the previously listed competitors. In particular, Table 4.7 lists the bugs found by LIBERATOR. We analyze crashes found when testing the target libraries by using the best $t_{\text{gen}}/t_{\text{test}}$ configuration from §4.6.3. We denote (with †) additional bugs discovered during LIBERATOR development. Later, we discuss the false

Table 4.4 – Library coverage after 24 hours fuzzing campaign for LIBERATOR, OSS-Fuzz, and HOPPER. LIBERATOR results are shown for t_{gen} of 24-, 18-, 12-, and 6-hours, with **bold** for the best configuration. Additionally, we report the coverage delta between the best LIBERATOR configuration and the other tools in the columns marked with Δ. **Bold green** highlights where LIBERATOR prevails. We mark unsupported libraries with a dash.

Target	LIBERATOR [t_{gen} + t_{test}]				HOPPER		OSS-Fuzz	
	24h+0h	18h+6h	12h+12h	6h+18h	cov.	Δ	cov.	Δ
c-ares	**55.29%**	50.88%	49.06%	52.65%	60.43%	-5.15%	22.62%	**+32.66%**
cJSON	**75.34%**	73.20%	71.52%	72.68%	87.44%	-12.10%	45.95%	**+29.39%**
cpu_features	**19.22%**	**19.22%**	**19.22%**	**19.22%**	18.06%	**+1.16%**	-	-
libaom	10.98%	10.57%	5.51%	**11.77%**	9.79%	**+1.98%**	44.20%	-32.43%
libdwarf	**18.08%**	17.88%	17.89%	17.89%	11.56%	**+6.51%**	20.04%	-1.96%
LibHTP	**26.74%**	25.07%	16.34%	17.37%	44.14%	-17.41%	31.96%	-5.22%
libpcap	**40.93%**	40.81%	37.48%	38.03%	36.15%	**+4.78%**	43.14%	-2.20%
libplist	54.30%	**54.43%**	53.97%	51.13%	63.07%	-8.63%	35.34%	**+19.10%**
libsndfile	26.50%	31.75%	35.46%	**36.59%**	6.18%	**+30.41%**	11.14%	**+25.45%**
LibTIFF	24.45%	26.93%	**27.70%**	26.36%	-	-	28.33%	-0.62%
libucl	56.34%	57.02%	57.43%	**57.78%**	64.98%	-7.20%	36.88%	**+20.90%**
libvpx	8.24%	**8.90%**	3.71%	7.59%	6.18%	**+2.71%**	48.48%	-39.58%
minijail	**18.45%**	15.48%	16.03%	16.07%	-	-	-	-
pthreadpool	**50.44%**	44.21%	36.42%	40.42%	31.01%	**+19.43%**	-	-
zlib	58.32%	**58.83%**	46.26%	55.01%	38.92%	**+19.91%**	50.14%	**+8.69%**

positives generated by LIBERATOR and their root causes. Finally, we expand on two case studies about libpcap and cJSON to illustrate the effectiveness of LIBERATOR in finding real-world bugs.

To correctly classify the bugs, we manually inspect their root cause and automatically cluster them in coherent classes via stack similarities [105]. Overall, LIBERATOR identifies 81 unique crashes across all 15 libraries. After triage, we report 24 confirmed bugs resulting in a 25% true positive ratio which is double that of similar state-of-the-art works [60]. HOPPER finds only six bugs, while the drivers from OSS-Fuzz, OSS-Fuzz-Gen, FuzzGen, and UTOPIA found *zero* during our evaluation.

True positives: Table 4.7 list the 24 bugs found by LIBERATOR. Each bug was promptly reported to the maintainers with a fix suggestion. LIBERATOR finds a variety of bugs, including logic errors, NULL dereferences, and integer overflows across six libraries. For example, at two locations, LibTIFF used to index arrays with signed integers, allowing a negative value to deceive the bounds check. This value would later be interpreted as an unsigned integer, leading to an index out of bound. The logic error in cJSON results from incorrect usage of strcpy from the C standard library. In libucl, LIBERATOR triggered seven crashes due to incomplete type checks, and an edge case where a non NULL-terminated string caused an out-of-bound read. Overall, the bugs identified ranged from shallow (*e.g.,* a single API call necessary) to complex (over three

Table 4.5 – LIBERATOR comparison against UTOPIA. The libraries used are from the commits indicated in UTOPIA. LIBERATOR's configurations are from Table 4.4.

Target	UTOPIA	LIBERATOR *	
	cov.	cov.	t_{gen} + t_{test}
cpu_features	4.45%	**54.97%**	24h + 0h
libaom	**18.66%**	15.61%	6h + 18h
LibHTP	**40.37%**	27.02%	24h + 0h
libvpx	8.41%	**9.31%**	18h + 6h
minijail	**31.59%**	18.73%	24h + 0h
pthreadpool	35.19%	**44.63%**	24h + 0h

Table 4.6 – LIBERATOR comparison against FuzzGen, and OSS-Fuzz-Gen. LIBERATOR's configurations are from Table 4.4.

Target	FuzzGen	LIBERATOR	
	cov.	cov.	t_{gen} + t_{test}
libaom	0.11%	**11.77%**	6h + 18h
libvpx	**9.35%**	8.90%	18h + 6h
	OSS-Fuzz-Gen	LIBERATOR	
cJSON	58.29%	**75.34%**	24h + 0h
libdwarf	**18.66%**	18.08%	24h + 0h
libpcap	1.09%	**40.93%**	24h + 0h
libplist	26.18%	**54.43%**	18h + 6h
libsndfile	0.58%	**36.59%**	6h + 18h
libucl	33.30%	**57.78%**	6h + 18h

chained API calls with inter-procedural dependencies). LIBERATOR can adapt to these different scenarios, fuzzing deeper once the shallow API functions are covered.

During our evaluation, HOPPER identifies 894 unique crashes but only *six* are true positives. The cause of HOPPER's high false positive rate is the imprecision of its sanitizer. Additionally, HOPPER's oracle overlooks the object lifecycle leading to even more false negatives. Two true positives were in `pthreadpool` and four in `libucl`, all of which were also identified by LIBERATOR.

Our fuzzing of the OSS-Fuzz drivers leads to *zero* crashes, likely due to the exhaustive testing that these drivers already experienced as part of the continuous campaigns lead by Google. This highlights the need for a broader and continuously evolving set of drivers. The bugs found by LIBERATOR prove that OSS-Fuzz drivers are not exhaustive and that LIBERATOR complements these harnesses leading to the discovery of new bugs.

UTOPIA, OSS-Fuzz-Gen, and FuzzGen fail to produce any true positive during our experiments. As we were unable to generate drivers for both OSS-Fuzz-Gen and FuzzGen, the publicly available harnesses have likely already been extensively tested. Notably, the OSS-Fuzz-Gen drivers for `cJSON` specifically target an API function—`cJSON_duplicate`—in which LIBERATOR found a bug. However, OSS-Fuzz-Gen's drivers are unable to trigger it because they all exercise the same sequence of API functions commonly used in consumers while LIBERATOR finds alternative sequences. To further showcase the effectiveness of LIBERATOR, we empirically confirm its capability to find the two bugs identified by OSS-Fuzz-Gen in previous versions of `cJSON` and `libplist`.[1] The drivers generated by UTOPIA, despite their good coverage during our evalua-

[1] Two out-of-bound access bugs: issues #800 in `cJSON` and #244 in `libplist`.

Table 4.7 – True positive found by LIBERATOR. The table reports the library and the bug type. The † indicates that the bug was found during development (§4.6.4). The status column indicates if the bug is fixed or only acknowledged by the maintainers. For reference, we report the project issue number. The NULL dereference in `libpcap` is tracked under CVE-2024-8006.

Library	Bug Type	Status	Issue/PR #
`cJSON`	Logic error	Fixed	*881*
`cJSON`	NULL deref.	Fixed	*882*
`cJSON`	Stack exhaust.	Fixed	*880*
`cJSON`	Stack exhaust.	Fixed	*880*
`cJSON`	Stack exhaust. †	Ack.	*827*
`cJSON`	Logic error †	Fixed	*821*
`libpcap`	NULL deref.	Fixed	*1345*
`LibTIFF`	SEGFault	Fixed	*643*
`LibTIFF`	Int. Overflow	Fixed	*649*
`LibTIFF`	Int. Overflow	Fixed	*645*
`LibTIFF`	Arith. Except.	Fixed	*646*
`LibTIFF`	Arith. Except. †	Fixed	*580*
`LibTIFF`	Arith. Except. †	Fixed	*628*
`libucl`	Miss. Check	Fixed	*315*
`libucl`	Miss. Check	Fixed	*315*
`libucl`	Miss. Check	Fixed	*315*
`libucl`	Miss. Check	Fixed	*315*
`libucl`	Miss. Check	Fixed	*315*
`libucl`	Miss. Check	Fixed	*315*
`libucl`	Miss. Check	Fixed	*315*
`libucl`	OOB Read	Ack.	*316*
`pthreadpool`	Arith. Except.	Fixed	*46*
`pthreadpool`	Heap Overflow	Ack.	*47*
`zlib`	Logic error †	Fixed	*889*

Table 4.8 – Breakdown of false positives generated by LIBERATOR. We left out libraries without false positives, namely `cpu_features`, `libplist`, and `libvpx`. *incoherent arg.* indicates incoherency between two or more arguments (*e.g.,* missed *Var-len* relationship). *mem. leak* denotes a memory leak. *use-after-free* points to a `free` incorrectly tracked. *malloc arg.* indicates unbounded size in a call to `malloc`. *non init buffer* occurs when uninitialized buffers are used. Finally, *API usage* conveys deviation from the intended usage.

	incoherent arg.	memory-leak	use-after-free	malloc arg.	non-init buffer	API usage	**Total**
`c-ares`	3	1	1	0	0	0	5
`cJSON`	1	0	0	0	1	0	2
`libaom`	1	0	0	0	0	0	1
`libdwarf`	7	1	1	0	2	1	12
`LibHTP`	2	0	2	0	0	0	4
`libpcap`	0	0	0	1	1	0	2
`libsndfile`	6	1	2	0	0	0	9
`LibTIFF`	0	0	0	0	2	0	2
`libucl`	2	1	1	2	1	0	7
`minijail`	2	1	0	0	0	1	4
`pthreadpool`	1	0	0	1	0	0	2
`zlib`	2	2	0	1	0	2	7
Total	27	7	7	5	7	4	57

tion, produce only false positive highlighting the limitation of basing the generation on existing test suites.

In addition to fixes for these bugs, some libErator drivers were also adapted as test cases for the libraries. For example, we contributed *two* test cases for the `cJSON` library. Following the maintainers' demand, we also contributed libErator drivers to `libpcap` for integration into their fuzzing campaigns. This demonstrates the capacity of libErator to produce valid and interesting drivers, complementary to the existing manual efforts of the maintainers.

libErator exhibits a high true positive ratio, *e.g.,* much higher than what we observed for Hopper and double the one reported by UTopia, and finds real-world bugs in a variety of thoroughly tested libraries, demonstrating its applicability.

False positives: We present libErator's false positives in Table 4.8. They can be grouped into two broad categories: (a) incoherent arguments (*e.g.,* when libErator misses a *Var-len* dependency), and (b) incorrect handling of memory (*e.g.,* missed *Sink* leading to Use-After-Free or unbounded size in `malloc`). These false positives are caused by the imprecision of our static analysis. For example, libErator currently misses the dependency between 2D arrays and their dimensions (*e.g.,* width and height of an image) leading to unwanted out-of-bound accesses. Additional engineering effort could avoid these false positives.

Only 7% of libErator false positives are caused by drivers using the library incorrectly. For example, in `zlib`, the documentation states that `inflateReset` should not be called after `deflateInit_`. However, libErator is not able to infer this incompatibility. Understanding such dependencies would require more complex static analysis or information external to the source code (*e.g.,* annotations or documentation). We leave this as future work.

Case study: `cJSON` is an lightweight JSON parser. In addition to straight forward NULL dereference, libErator found logic bugs, for example the function `cJSON_SetValuestring` could pass overlapping strings to `strcpy`, which is explicitly disallowed. The maintainers acknowledged and fixed this bug. libErator also find multiple stack exhaustion in `cJSON` caused by incorrect handling of circular references in JSON objects. Lastly, libErator identifies inconsistencies in how `cJSON` handles arrays and dictionaries. The maintainers acknowledged the bug and are reflecting on how to address it as a fix would break the API compatibility. libErator's `cJSON` false positives are caused by two missing *Var-len* relationships and missing the identification of a *Sink* function resulting in a freed variable being passed to an API function triggering a Use-After-Free.

Case study: `libpcap` is a widely used library for packet capture. It is continuously fuzzed through three manually written fuzz drivers as part of the OSS-Fuzz [109] project. Nonetheless, libErator generated novel drivers that resulted in three crash clusters. Both false positives were caused by allocating buffers smaller than the fixed size mandated in the documentation. The last crash was a NULL dereference in `pcap_findalldevs_ex` when the directory passed is

Table 4.9 – Library coverage for a t_{gen} of 24 hours across both *full* LIBERATOR and LIBERATOR without field bias. In the last column, we report the difference between the two, highlighting in **bold green** when LIBERATOR prevails.

Target	LIBERATOR w/o field bias	*full* LIBERATOR	Δ
c-ares	55.48%	55.29%	-0.19 %
cJSON	74.62%	75.34%	**0.71 %**
cpu_features	19.22%	19.22%	**0.00 %**
libaom	8.37 %	10.98%	**2.61 %**
libdwarf	18.15%	18.08%	-0.07 %
LibHTP	10.47%	26.74%	**16.26%**
libpcap	42.25%	40.93%	-1.32 %
libplist	52.60%	54.30%	**1.71%**
libsndfile	21.76%	26.50%	**4.74 %**
LibTIFF	20.65%	24.45%	**3.80 %**
libucl	53.53%	56.34%	**2.81 %**
libvpx	7.83 %	8.24 %	**0.41 %**
minijail	19.53%	18.45%	-1.07 %
pthreadpool	47.96%	50.44%	**2.48 %**
zlib	61.27%	58.32%	-2.95 %

Table 4.10 – The first two columns report the number of drivers generated in 24 hours and how many are selected for fuzzing. The last column shows the duration of this process. Driver selection drastically reduces the number of drivers to test while requiring negligible resources.

Target	Tot. drv	Sel. drv	Avg [s]
c-ares	746	50	0.71
cJSON	1343	102	5.09
cpu_features	2046	7	13.14
libaom	1323	85	3.15
libdwarf	1173	101	3.09
LibHTP	1476	110	6.49
libpcap	931	57	1.72
libplist	2404	25	0.10
libsndfile	970	81	1.76
LibTIFF	612	45	0.81
libucl	910	64	1.06
libvpx	1624	86	7.40
minijail	300	19	5.61
pthreadpool	255	22	0.70
zlib	948	73	3.46

non-existent. The maintainers acknowledged the bug and awarded CVE-2024-8006. We provided a fix that is now upstream. The OSS-Fuzz drivers were not able to find this bug because they are limited to a narrow subset of the whole exposed API and do not test `pcap_findalldevs_ex`. Following our finding, the `libpcap` maintainers requested integration of LIBERATOR drivers into their fuzzing campaign.

4.6.5 RQ5 - Ablation Study

We conduct two ablation studies to understand how the components of LIBERATOR contribute to its results. First, we evaluate the impact of the AFG bias used to guide the API Chain creation (§4.4.1) and to solve challenge C1. The second study aims at understanding the impact of the clustering developed to avoid fuzzing redundant drivers (§4.4.3), *i.e.,* challenge C3, on the overall performance of LIBERATOR. Regarding challenge C2, an experiment excluding this technique is infeasible since it is core to LIBERATOR's driver generation, and removing it would distort the whole architecture.

Table 4.9 shows the coverage reached after 24 hours of driver generation (24 hours t_{gen}) by two libErator configurations. First, we run libErator without biasing the AFG. In practice, we modify the function `random_bias` (line 18 and 21 in algorithm 2) to pick fully at random an API function out of the possible candidate. The AFG bias shows crucial improvements in the coverage of `LibHTP`. The complex chains necessary to set up minimal state in `LibHTP` are unlikely to be encountered by libErator without biasing the choice of the subsequent API function during driver generation due to the size of the API of `LibHTP`. Coverage-guided driver generation (*e.g.,* Hopper) shows similar limitations due to the limited coverage reached by their API Chains. For the other libraries, AFG bias does not show a particular effect. From our analysis, this is because most API functions interact with a similar number of fields, therefore, reducing the bias effect.

To assess the impact of our driver selection strategy (§4.4.3), we measure the duration of the affinity propagation clustering across the drivers generated by libErator in 24 hours with a similar setup as the previous study. Results are presented in Table 4.10. Despite having to cluster more than 2'000 drivers, the duration is negligible for all the libraries. In the worst case, `cpu_features` takes less than 0.02% of the total experiment time. This shows that the driver selection is not a bottleneck in the pipeline of libErator. By reducing the number of drivers by up to 99%, the driver selection is crucial to avoid redundant drivers and distribute the fuzzing resources efficiently.

Overall, each component of libErator contributes to the increased performance shown in §4.6.2. The AFG bias is crucial to guide the driver generation for libraries with complex setups (*e.g.,* `LibHTP`) while the driver selection is lightweight.

4.7 Discussion

Future work. So far, libErator cannot generate drivers containing loops. Supporting loop involves an explosion of complexity as libErator must decide which functions should be called repeatedly and how to handle loop termination. With engineering efforts, however, our static analysis could identify the API functions expected to be used in loops based on argument types.

Despite the design concepts being generic and language-agnostic (§4.4), our prototype targets C code, but we plan an extension to other languages, such as C++ or Python. For example, libErator for C++ could reuse most of the architecture. New constructs like class hierarchies, templates and generics are not yet handled by libErator resulting in an incomplete AFG for C++ code.

Our current static analysis (§4.4.1) inherits SVF imprecision when following indirect jumps. Ongoing efforts to improve SVF point-to analysis would enhance the AFG (*e.g.,* avoiding the false positives due to missed *Var-len*). Lastly, SVF cannot analyze some libraries due to state explosion.

Comprehensively understanding the different library characteristics influencing its position along the t_{gen} versus t_{test} ratio remains an open question.

Limitations. libErator adds empty stub functions when an API requires function pointers. Understanding which function behavior is expected is a daunting task that would require other sources of information (*e.g.,* consumers or documentation). Additionally, to further increase the probability of synthesizing valid API Chains, we could leverage Large Language Model (LLM). We consider these techniques orthogonal to ours since we aim to show the limitations of one-size-fits-all solutions and the complexity of striking a balance between driver generation and deeper testing.

4.8 Related Work

Automated library testing. Randoop [94] and EvoSuite [35] pioneered the automatic generation of tests for Java programs. The latter is based on test case mutations and invariants inference to define specifications that can later be verified. Similar work exist for other languages [82, 88]. Documentation and source code comments have also been used to discover software specifications [10]. Lately, LLMs have improved test suites for functions with little coverage [72]. Contrary to these works, libErator generates fuzzing drivers unleashing fuzzing engine on C libraries.

Driver synthesis. FuzzGen [56], Fudge [5], UTopia [60], OSS-Fuzz-Gen [54], PromptFuzz [132], Winnie [63], AFGen [75], IntelliGen [138], TitanFuzz [28], RUBICK [136], Daisy [139], and NEXZZER [**nexzzer**] leverage external knowledge about the API usage (*consumer-dependent*). Either by trimming and cross-pollinating consumers, transforming unit tests, or by querying LLMs, these tools learns from existing usage and are limited in the diversity of the generated drivers. Moreover, as exemplified in APICraft, they focus on driver generation while not exploring trade-offs between generation and testing, and overlook driver selection strategies. Conversely, libErator separates the duty of generating and testing fuzz drivers, showing that the two problems are orthogonal and a single *one-for-all* solution is not achievable as that strongly depends on the library API.

Hopper [18] and GraphFuzz [47] model library testing as a grammar fuzzing problem and, as libErator, are consumer-agnostic. However, both fail to address the challenges in §4.2. Conversely, libErator helps strike a balance specific to each library, addressing a new angle of this problem.

4.9 libErator summary

We laid out the computation trade-off between the time spent generating drivers and the resources invested in fuzzing, and limitation it imposes to the current library fuzzing works. We highlighted the necessity for a driver generation technique capable of dividing resources differently for each library, striking a better balance for each library than a one-size-fits-all approach.

We presented libErator, a novel methodology, configurable along this trade-off, to automatically generate fuzz drivers. libErator employs cutting-edge static analysis techniques to infer the semantics of API functions and, then, generate valid fuzz drivers for a target library. We deploy libErator against 15 libraries and compare our results against both the state-of-the-art driver synthesis generators and manually written drivers.

libErator reaches more coverage than other automated tools while finding bugs in extensively tested libraries where manual written drivers find none. We found 24 bugs that were reported to the maintainers. We upstream derived test cases and fuzz drivers to multiple projects.

Chapter 5

Future work

This thesis advances the state-of-the-art in securing system code, yet several research directions remain to be explored. Beyond the necessary adoption efforts, we identify key areas for future investigation.

5.1 Type confusion detection and mitigation

While our work focused on preventing derived type confusions and detecting unrelated ones, several challenges remain. First, extending Sourcerer to verify casts to integral types—*e.g.,* from `A*` to generic pointers like `void*`—at a minimal overhead allowing for its deployment in adversarial scenarios. One promising approach involves identifying and replacing the would-be superfluous developer-implemented type systems with standard RTTI checks.

Our fuzzing campaign revealed limitations stemming from corrupted memory layouts after unrelated type confusions. The inline storage of type metadata means that type confusions often corrupt the other fields with the RTTI content. A potential solution involves storing type information in an adjoint metadata area for each allocation site, similar to ASan red zones [106]. While this would increase overhead for standalone deployments—ASan experiences a 70% overhead on average—the cost would be marginal when combined with existing ASan instrumentation—a common scenario in fuzzing campaigns.

Lastly, most of the type confusions reported nowadays in modern browsers are in JavaScript code and not in C++. Extending type confusion detection and mitigation to type-unsafe languages, especially in a cross-language environment, is a promising venue. Due to the dynamic typing of JavaScript, new strategies are required to track types throughout execution.

5.2 Library fuzzing

Despite widespread industry adoption through projects like OSS-Fuzz [109], library fuzzing struggles both in terms of driver quality and quantity. For instance, only four out of the nine OpenCV [12] fuzz drivers contribute meaningful coverage beyond the empty seed—even after

24 hours. While libErator demonstrates that automated tools can generate effective drivers at scale, triaging the resulting crashes still requires understanding the driver's content. Future work should focus on generating *explainable* drivers to facilitate both triaging between false and true positive as well the upstreaming long-term maintenance of the drivers.

libErator's key insight about the trade-off between driver generation and testing time opens new research questions. Understanding the factors influencing this balance would help library developers select the right balance without a lengthy optimization journey. This includes investigating how library characteristics such as API complexity, affect the optimal allocation of resources between driver generation and execution.

Chapter 6

Data Availability

6.1 type++

type++ prototype implementation is available under the Apache-2.0 Licence at https://github.com/HexHive/typepp. The artifact-evaluated version is also available from persistent storage at DOI:10.5281/zenodo.13687049. The detailed instructions on how to run our artifact are available in Appendix B.

6.2 Sourcerer

Our implementation of Sourcerer is available on GitHub: https://github.com/HexHive/Sourcerer. We additionally provide the script for the evaluation on the same repository.

6.3 libErator

Our code is on GitHub and Zenodo with DOI:10.5281/zenodo.15201791 [121] under the Apache-2.0 license. Due to length of the fuzzing campaign, the coverage data is too large for Zenodo. We, however, release the raw aggregated results as `CSV`. The instructions on how to run the experiments are in the main GitHub repository and in Appendix D.

Chapter 7

Conclusion

This thesis advances the safety of low-level programming languages through three complementary contributions targeting type safety and automated testing. As C and C++ continue to underpin critical software infrastructure, securing these codebases is paramount. Our solutions focus on minimizing the developer effort and practical deployment, recognizing the challenges of maintaining large codebases with limited resources.

Our first contribution, type++, provides a novel C++ dialect that prevents derived type confusion by design. By inlining type metadata in objects, we achieve comprehensive runtime type checking for downcasts with minimal overhead ($< 1\%$) thanks to tracking fewer lifetime changes. The approach requires minimal code adaptations (0.04% LoC in the SPEC CPU benchmarks) while protecting 23×more casts than the state-of-the-art solutions. Our evaluation revealed 122 type confusion bugs, including 14 previously unknown vulnerabilities.

Building on insights from type++, our second contribution, Sourcerer, extends type confusion detection coverage to *all* casts, including unrelated type conversions. Through a specialized type information initializer (RTTIInit), we improve compatibility with C++ while supporting previously missed edge cases like unions. With only 5% runtime overhead, Sourcerer doubles the number of verified unrelated casts compared to type++. Notably, we conducted the first fuzzing campaign specifically targeting type confusions, discovering six new type confusion vulnerabilities in OpenCV.

Our third contribution, LIBERATOR, addresses the challenge of automated library testing. By generating valid fuzz drivers without requiring consumer code and minimal developer involvement, we enable efficient testing of C libraries. Our approach uniquely balances driver generation and testing time, discovering bugs in well-tested libraries where manual-written drivers and other automated approaches fail to discover any.

Together, these contributions demonstrate how automated approaches and tackling issues at the language level can significantly improve software security while minimizing the developer burden. While transitioning to memory-safe languages like Rust offers long-term security benefits, our solutions provide immediate, practical improvements for existing C/C++ codebases.

Appendices

Appendix A

type++ Manuscript

We present interesting use cases and an experiment demonstrating the capabilities of type++ as a sanitizer during fuzzing.

A.1 type++ as sanitizer

We conduct a preliminary analysis to evaluate type++ as a lightweight sanitizer in a fuzzing campaign. For this experiment, we test V8 [45] version 9.0.257.29, the Chromium JavaScript engine. We compile V8 in two configurations: one applying Property 3 and protecting a random set of 32 classes (named *typepp*), and a second one with the default V8 fuzzing configuration (named *vanilla*). FUZZILLI [48], a mature JavaScript fuzzing tool compatible with V8 (version 0.9.3) serves as our fuzzer. The experiments were executed on an Intel machine with an i7-8700 @ 3.20GHz CPU and 64GB of RAM. The fuzzing campaign lasted for 12h, starting with an empty corpus. The *vanilla* configuration reaches 15% of branches as coverage, while the *typepp* one 8% of the branches. The difference in coverage is caused by new type confusion bugs detected by type++ that block the exploration of some V8 components. These bugs were not found by ASan [106] (the default sanitizer in the *vanilla* configuration), which cannot identify type confusion anomalies. ASan would only crash if a type confusion results in a later memory safety violation. We are triaging these issues together with the Chromium team. This experiment demonstrates that type++ can be used in a fuzzing campaign as an alternative sanitizer in combination with other tools such as ASan.

A.2 Use case: POV-Ray

POV-Ray [98] is a ray-tracing program that generates images from a text-based scene description. For POV-Ray, in both SPEC CPU2006 and CPU2017, we found improper use of C-style `structs` to simulate parent-child relationships. Specifically, POV-Ray contained a set of `structs` implemented as unrelated types (*i.e.,* they did not declare a parent class) but that were used as parent-child classes. The program casts those classes in a C-style fashion assuming that the memory layout of the two classes overlaps. Blindly casting two structures is undefined behavior since the compiler

```
1 class DOMTextImpl: public DOMNode {
2   DOMNodeImpl fNode;
3   ...
4 };
5
6 class DOMElementImpl: public DOMNode {
7 public:
8   DOMNodeImpl fNode;
9   ...
10 };
11
12 // DOMTextImpl at runtime
13 DOMNodeImpl *castToNodeImpl(const DOMNode *p) {
14   DOMElementImpl *pE = (DOMElementImpl *)p;
15   // works because the first
16   // element is fNode
17   return &(pE->fNode);
18 }
```

Listing A.1 – A (simplified) type confusion example in Xalan-C++ from SPEC CPU2006.

might apply an optimization over one `struct` layout. Therefore, we consider these C-style casts as bugs. When running, type++ immediately identified and reported the type confusion occurrences, we thus identified 113 program locations for SPEC CPU2006 and 99 program locations for SPEC CPU2017 that triggered a type confusion bug. We then used the information from type++ to infer the correct class hierarchy and modify the source code accordingly. In POV-Ray, we edited 123 LoC in SPEC CPU2006 and 131 LoC in SPEC CPU2017, corresponding to around of their codebase. All the modifications concerned the correction of 19 classes in which we included the proper parent class in the header files, while the program logic remained unchanged. These changes were necessary as the LLVM cast checks were crashing due to non-existent RTTI, highlighting the type confusion. Modifications to allow the cast check from LLVM to recover are left as future work.

A.3 Use case: Xalan-C++

Xalan-C++ is an XSLT processor originally developed by IBM [2]. The program is written in complex C++ and heavily overloads the class hierarchy to represent the DOM of an XML document. Moreover, Xalan-C++ implements a sophisticated memory allocation strategy based on `new()` primitives and `operator::new()` overloads to optimize the allocated memory. This section focuses on the SPEC CPU2006 version, but the same reasoning applies to SPEC CPU2017. We managed to compile Xalan-C++ with type++ and run the benchmark, type++ (as well as LLVM-CFI) reported around 9,000 type confusion errors at seven unique locations. The type confusions were generated because Xalan-C++ was attempting to cast unrelated objects that shared

part of their layout (similar to Blender and POV-Ray). Listing A.1 contains an example of type confusion detected: `castToNodeImpl()` gets a `DINNode` object as an input and tries to cast it into a `DOMElementImpl` type. At runtime, the function receives a `DOMTextImpl` object, which is semantically correct since it is a child of `DOMNode`. However, even though the classes share the same parent, they belong to different branches in the class hierarchy, thus creating type confusion. Nonetheless, the code works because the two objects have `fNode` as a common first field.

Our proposed solution would use multiple inheritances, in particular, `castToNodeImpl()` should accept a new type, *e.g.,* `DOMConverted`, that is added as an additional parent to `DOMTextImpl` and `DOMElementImpl`. Moreover, `DOMConverted` should implement a virtual function, *e.g.,* `getNode()`, that is implemented in the subclasses (*i.e.,* `DOMTextImpl` and `DOMElementImpl`) and returns the field `fNode` without relying on brittle memory layout assumptions. Adopting this technique would allow Xalan-C++ to avoid type confusions. Since the modification requires deep code modification (due to the intertwined relationship with the custom allocators), we leave patching these issues for future work. In this case, if the code base is too old, it is possible to have a trade-off between security and usability by leaving only 7 locations as possible attacker-surface and exempting those from runtime checking.

Appendix B

type++ Artifact

In this appendix, we provide the requirements, instructions, and further details necessary to reproduce the experiments from our paper (DOI: 10.14722/ndss.2025.230053).

B.1 Description and requirements

The artifact contains the material to reproduce the results: Compatibility analysis (§2.5.1 – Table 2.1), Performance Overhead & Security evaluation (§2.5.2 & §2.5.3 – Table 2.2 & Table 2.5), Use case: Chromium (§2.5.4 – Table 2.6). This material is released under the Apache License 2.0, in line with the LLVM project type++ builds upon.

1. *Accessing the artifact*: We release the artifact on a public GitHub repository: `https://github.com/HexHive/typepp`. While the typepp branch contains the latest version of the code, the `ndss-25-artifacts` tag contains the exact version of the code that was submitted for review in the artifact evaluation. Additionally, the code is available on Zenodo with the DOI:10.5281/zenodo.13687049.
2. *Hardware dependencies*: The artifact requires a machine with at least 128GB of RAM and a 1TB disk. 16GB of RAM is sufficient to run the SPEC CPU evaluations.
3. *Software dependencies*: The artifact was tested on Ubuntu 20.04 and require the ability to run Docker containers. An active internet connection is also necessary for the Chromium evaluation. Additionally, `curl`, `git`, `docker`, and `pip` should also be installed.
4. *Benchmarks*: The artifact requires a copy of the SPEC CPU 2006 & 2017 benchmarks.[1]

B.2 Artifact installation

The initial step is to clone the repository and build the Docker image. As the artifact is provided on top of a fork of the LLVM project, we recommend to only proceed to a shallow

[1] `https://www.spec.org/cpu2006/` and `https://www.spec.org/cpu2017/`

clone of the last 100 commits. This can be achieved by running the following bash command: `git clone $REPO --single-branch --branch typepp --depth 100 LLVM-typepp`. Additionally, please run `pip install -r requirements.txt` from inside the repo to install dependencies.

Each experiment is encapsulated in one or multiple Docker container. The Dockerfile is available at the root of the artifact repository. We do not provide support for running the experiments locally.

B.3 Experiment workflow

Our artifact aims at reproducing the results from three experiments presented in the paper. The first aims at evaluating the compatibility of type++ with existing software and quantifies the number of LoC changes necessary to port a C++ project. The second experiment evaluates the performance overhead of type++ over standard C++ with the help of the SPEC CPU 2006 and 2017 benchmarks as well as quantifies the added security guarantees provided by type++. Lastly, the third experiment demonstrate the performance and security benefits of type++ after a partial support of the Chromium browser.

We propose to run this experiments sequentially as they are presented in the paper. The artifact provides scripts to run the experiments and collect the results. The scripts will also generate tables similar to the ones presented in the paper.

More complete and detailed instructions, as well a minimal example, are available in the README file of the repository. We highly recommend following the instructions there as copying and pasting the commands from this document might introduce errors.

B.4 Major claims

- (C1) *Compatibility*: type++ is compatible with existing C++ codebases with a few minor changes. This is showcased in experiment E1 which runs our compatibility analysis on the SPEC CPU benchmarks. These results are presented in Table 2.1 in the paper.
- (C2) *Performance Overhead & Security Guarantees*: type++ incurs a negligible performance while protecting a vast amount of additional casts. Our experiment E2 highlights this trend on the SPEC CPU benchmarks. In the paper, the results are available in Table 2.2. The overhead numbers can slightly differ from the ones in the paper due to the different hardware configurations, but we expect the global trend across the benchmark to remain consistent.

Due to time constraints, we do not provide automatic scripts to run and extract the data from our competitors (i.e., HexType, TypeSan, and EffectiveSan) that would be necessary for Table 2.3. We, however, provide instructions on how to run these experiments.[2]

- (C3) *Type checking cost breakdown*: To better assess the cost of the different type checking methods, we designed a micro-benchmark. The results are presented in Table 2.5 and can be reproduced through the experiment E3.

- (C4) *Use case: Chromium*: type++ can be used to protect large code bases. As a proof of concept, we partially protect the Chromium browser. The results are presented in Table 2.6 and can be reproduced through the experiment E4.

B.5 Evaluation

In this section, we provide the detailed steps to run the experiments and process the results to get the tables presented in the paper. Overall, this process requires around two to three days of computation time on a powerful server. These instructions are also available in the README file of the artifact repository.

Experiment 1 (E1) - Claim (C1): Compatibility Analysis:

[2 humans minutes + 2 compute-hours] The experiment evaluates the porting effort necessary for SPEC CPU 2006 and 2017 benchmarks and quantify the benefits of Property 2 over Property 1. The experiment consists of compiling the benchmarks with type++ to first collect the classes to instrument for Property 2 and then run the analysis to collect the warnings emitted for both properties.

[Preparation] Ensure that the two SPEC CPU benchmarks `.iso` are available at the root of the cloned repository.

[Execution] Run the commands:

```
1 # Build the docker images and then run two analysis (Property 1 and Property 2) on
  ↪  the SPEC CPU benchmarks (around 2hours).
2 # Expected output: Different logs highlighting first the Docker builds and then the
  ↪  benchmarks compilation and analysis. Finally, a table similar to Table I will be
  ↪  printed.
3 ./table1.sh
```

[2] `https://github.com/HexHive/typepp/blob/typepp/COMPETITORS.md`

[Results] Upon completion, the script will generate a table identical to Table I. The file `analysis_result_test.tex` contains the results.

Experiment 2 (E2) - Claim (C2): Performance Overhead & Security Guarantees: [2 humans minutes + 15 compute-hours] This experiment runs the SPEC CPU benchmarks with different flavors of cast checking. First, a baseline is established by running the benchmarks with no cast checking. Then, the benchmarks are run with type++ and LLVM-CFI to measure the overhead. Lastly, an extra run for both type++ and LLVM-CFI is performed to measure the number of cast protected.

[Preparation] Again, ensure that the two SPEC CPU benchmarks '.iso' are available at the root of the cloned repository.

[Execution] In a shell, run the following command:

```
# Build the docker images and then run all the benchmarks variation (around
  ↪ 15hours).
# Expected output: Different logs highlighting first the Docker builds and then the
  ↪ benchmarks compilation and execution. Finally, a table similar to Table II will
  ↪ be printed.
./table2.sh
```

[Results] Upon completion, the script will generate a table similar to Table 2.1. There might be variations up to 10% in the performance and memory overhead numbers due to the different underlying machine. In particular, in a shared environment, the overhead might show inconsistencies. The number of casts protected should remain identical to the one presented in the paper.

Experiment 3 (E3) - Claim (C3): Type checking cost breakdown: [2 humans minutes + 20 compute-minutes] This experiment evaluates the cost of the different operations in the type checking process of HexType and type++.

[Preparation] No specific preparation is required.

[Execution] In a shell, run the following command:

```
# Build the docker images and then run the micro-benchmark (around 20minutes).
# Expected output: Different logs highlighting first the Docker build and then the
  ↪ cost of the different operations.
./table5.sh
```

[Results] Upon completion, the script will generate a table similar to Table 2.5. The numbers presented should exhibit similar ratios to the ones in the paper.

Experiment 4 (E4) - Claim (C4): Use case: Chromium: [2 humans minutes + 36 compute-hours] This experiment evaluates the performance of Chromium with partial protection by type++ and LLVM-CFI. It also outputs the number of casts protected by both tools. This experiment will build Chromium first with no protection, then with LLVM-CFI, and finally with type++. Then, it will run the JetStream benchmark and, therefore, require a working internet connection. As building and linking Chromium is a time-consuming and resource-intensive task, we recommend running this experiment on a machine with plenty of RAM and in `tmux` or `screen` session to avoid interruptions.

[Preparation] Please run the following script:

```
# Fetch the Chromium source code and the modifications we applied to it.
./table6_requirements.sh
```

[Execution] In a shell, run the following command:

```
# Build the docker images, inlcuding fetching the dependencies of Chromium, and then
  ↪ build Chromium itself. Lastly, run the JetStream benchmark (around 36hours).
# Expected output: Different logs highlighting first the Docker builds, the fetching
  ↪ of depencies and then the Chromium compilation and execution. Finally, a table
  ↪ similar to Table VI will be printed.
./table6.sh
```

[Results] Upon completion, the script will generate a table similar to Table 2.6. The benchmark scores presented should exhibit similar ratios to the ones in the paper. The number of casts protected is expected to be different as the Chromium execution is not deterministic but should remain in the same magnitude.

Appendix C

Sourcerer Artifact

In this Appendix, we provide the requirements, instructions, and further details necessary to reproduce the experiments from our paper `Sourcerer: channeling the void`.

C.1 Description and requirements

The artifact contains the material to reproduce the results: Porting Effort (Section 7.1 – Table 1), Performance Overhead (Section 7.2 – Table 2), Source of Performance Overhead (Section 7.3 – Table 3), and Sourcerer as a Sanitizer for Fuzzing Campaigns (Section 7.4 – Figure 1. This material is released under the Apache License 2.0, in line with the LLVM project Sourcerer builds upon.

1. *Accessing the artifact*: We release the artifact on a public GitHub repository. While the main branch contains the latest version of the code.
2. *Hardware dependencies*: The artifact requires a machine with at least 128GB of RAM and a 1TB disk. 16GB of RAM is sufficient to run the SPEC CPU evaluations.
3. *Software dependencies*: The artifact was tested on Ubuntu 20.04 and require the ability to run Docker containers. An active internet connection is also necessary for the Chromium evaluation. Additionally, `curl, git, docker`, and `pip` should also be installed.
4. *Benchmarks*: The artifact requires a copy of the SPEC CPU 2006 & 2017 benchmarks.[1]

C.2 Artifact installation

The initial step is to clone the repository and build the Docker image. As the artifact is provided on top of a fork of the LLVM project, we recommend to only proceed to a shallow clone of the last 100 commits. This can be achieved by running the following bash command: `git clone`

[1] `https://www.spec.org/cpu2006/` and `https://www.spec.org/cpu2017/`

`$REPO --single-branch --branch main --depth 100 Sourcerer`. Additionally, please run `pip install -r requirements.txt` from inside the repo to install dependencies.

Each experiment is encapsulated in one or multiple Docker container. The Dockerfile is available at the root of the artifact repository. We do not provide support for running the experiments locally.

C.3 Experiment workflow

Our artifact aims at reproducing the results from three experiments presented in the paper. The first aims at evaluating the compatibility of Sourcerer with existing software by quantifing the number of extra classes to instrument on top of type++. The second experiment evaluates the performance overhead of Sourcerer over standard C++ with the help of the SPEC CPU 2006 and 2017 benchmarks as well as quantifies the added security guarantees provided by Sourcerer. Then, we conduct an ablation study to understand where the overhead originate from. Lastly, the fourth experiment demonstrate the performance and security benefits of Sourcerer during a fuzzing campaign on OpenCV.

We propose to run this experiments sequentially as they are presented in the paper. The artifact provides scripts to run the experiments and collect the results. The scripts will also generate tables and figure similar to the ones presented in the paper.

More complete and detailed instructions, as well a minimal example, are available in the README file of the repository. We highly recommend following the instructions there as copying and pasting the commands from this document might introduce errors.

C.4 Major claims

- (C1) *Compatibility*: Sourcerer is compatible with existing C++ codebases with a few minor changes. This is showcased in experiment E1 which evaluates the number of extra classes to instrument as part of Sourcerer. These results are presented in Table 1 in the paper.

- (C2) *Performance Overhead*: Sourcerer incurs a negligible performance while protecting a vast amount of additional unrelated casts. Our experiment E2 highlights this trend on the SPEC CPU benchmarks. In the paper, the results are available in Table 2. The overhead numbers can slightly differ from the ones in the paper due to the different hardware configurations, but we expect the global trend across the benchmark to remain consistent.

- (C3) *Ablation Study of the Performance Overhead*: To better assess the cost of associated to the different component of Sourcerer, we conducted an ablation study. The results are presented in Table 3 and can be reproduced through the experiment E3.

- (C4) *Fuzzing Campaign*: Sourcerer can be used to test software project. As a proof of concept, we run a fuzzing campaign on the OpenCV project. The results are presented in Figure 1 and can be reproduced through the experiment E4.

C.5 Evaluation

In this section, we provide the detailed steps to run the experiments and process the results to get the tables presented in the paper. Overall, this process requires around two to three days of computation time on a powerful server. These instructions are also available in the README file of the artifact repository.

Experiment 1 (E1) - Claim (C1): Compatibility Analysis:

[2 humans minutes + 2 compute-hours] The experiment evaluates the extra number of classes necessary for SPEC CPU 2006 and 2017 benchmarks. The experiment consists of compiling the benchmarks with Sourcerer to first collect the classes to instrument.

[Preparation] Ensure that the two SPEC CPU benchmarks `.iso` are available at the root of the cloned repository.

[Execution] Run the commands:

```
# Build the docker images and then run two analysis (Property 1 and Property 2) on the SPEC CPU benchmarks (around 2hours).
# Expected output: Different logs highlighting first the Docker builds and then the benchmarks compilation and analysis. Finally, a table similar to Table 1 will be printed.
./table1.sh
```

[Results] Upon completion, the script will generate a table identical to Table 1.

Experiment 2 (E2) - Claim (C2): Performance Overhead [2 humans minutes + 15 compute-hours] This experiment runs the SPEC CPU benchmarks with different flavors of cast checking. First, a baseline is established by running the benchmarks with no cast checking. Then, the benchmarks are run with Sourcerer and LLVM-CFI to measure the overhead. Lastly, an extra run for both Sourcerer and LLVM-CFI is performed to measure the number of cast protected.

[Preparation] Again, ensure that the two SPEC CPU benchmarks '.iso' are available at the root of the cloned repository.

[Execution] In a shell, run the following command:

```
# Build the docker images and then run all the benchmarks variation (around 15hours).
# Expected output: Different logs highlighting first the Docker builds and then the benchmarks compilation and execution. Finally, a table similar to Table 2 will be printed.
./table2.sh
```

[Results] Upon completion, the script will generate a table similar to Table 2. There might be variations up to 10% in the performance and memory overhead numbers due to the different underlying machine. In particular, in a shared environment, the overhead might show inconsistencies. The number of casts protected should remain identical to the one presented in the paper.

Experiment 3 (E3) - Claim (C3): Ablation Study: [2 humans minutes + 15 compute-hours] This experiment evaluates the cost of the different operations in the type checking process of Sourcerer.

[Preparation] Again, ensure that the two SPEC CPU benchmarks '.iso' are available at the root of the cloned repository.

[Execution] In a shell, run the following command:

```
# Build the docker images and then run the micro-benchmark (around 15 hours).
# Expected output: Different logs highlighting first the Docker build and then the cost of the different operations.
./table3.sh
```

[Results] Upon completion, the script will generate a table similar to Table 3. The numbers presented should exhibit similar ratios to the ones in the paper.

Experiment 4 (E4) - Claim (C4): Fuzzing campaign: [2 humans minutes + 25 compute-hours] This experiment evaluates the fuzzing performance of two sanitizers: Sourcerer and ASan.

[Preparation] Please run the following script:

```
# Setup the fuzzing campaign
./fig1_requirements.sh
```

[Execution] In a shell, run the following command:

```
# Build the docker images
# Expected output: Different logs highlighting first the Docker builds, and finally,
  ↪  a figures similar to Figure 1 will be saved.
./fig1.sh
```

[Results] Upon completion, the script will generate figures similar to the ones in Figure 1. They are saved in the folder `fuzzing_pics`. The trends should be similar as in the paper.

Appendix D

libErator Artifact

In this Appendix, we provide the requirements, instructions, and further details necessary to reproduce the experiments from our paper *Liberating libraries through automated fuzz driver generation: Striking a Balance Without Consumer Code* (DOI: 10.1145/3729365).

D.1 Description and requirements

The artifact contains the material to reproduce the results: RQ1: t_{gen} vs t_{test} trade-off analysis (Section 7.1—Figure 3), RQ2: How does LIBERATOR test libraries? (Section 7.2—Figure 4 & Figure 5), RQ3: Comparison with state-of-the-art (Section 7.3—Table 4, LIBERATOR part), and RQ5: Ablation study (Section 7.5—Table 9 & Table 10). This material is released under the Apache License 2.0.

1. *Accessing the artifact*: We release the artifact on a public GitHub repository. While the main branch contains the latest version of the code, the `fse-25-artifact` tag contains the exact version of the code that was submitted for review in the artifact evaluation. Additionally, the code is available on Zenodo with the 10.5281/zenodo.13752276.
2. *Hardware dependencies*: The artifact requires a machine with at least 64GB of RAM and a 1TB disk.
3. *Software dependencies*: The artifact was tested on Ubuntu 20.04 and require the ability to run Docker containers. An active internet connection is also necessary for fetching the target libraries. Additionally, `curl`, `git`, `docker`, and `pip` should also be installed. We further provide a `preinstall.sh` script to install additional support software.

D.2 Artifact installation

The initial step is to clone the repository and build the Docker image. Additionally, please run `./preinstall.sh` from inside the repo to install dependencies. Notice that this script will also install and compile LLVM, this operation may take a few hours depending on your machine.

Each experiment is encapsulated in one or multiple Docker container. The Dockerfile is available at the root of the artifact repository. We do not provide support for running the experiments in the host machine.

D.3 Experiment workflow

Our artifact aims at reproducing the results from four experiments presented in the paper. The first aims at exposing the trade-off of testing vs creating drivers and compare our best results with our competitors. The second experiment evaluates of libErator drivers explore a library. Lastly, the third experiment demonstrate the value of each component of libErator by conducting an ablation study.

We propose to run these experiments sequentially as they are presented in the paper. The artifact provides scripts to run the experiments and collect the results. The scripts will also generate tables similar to the ones presented in the paper. Running the complete campaigns may require multiple days and around 10TB of free space. Due to the large amount of resources required to run the full experiments, we set a "quick" version of the experiments that take around 10 hours and require around 500GB of free space. This version ensures that the whole infrastructure works and the main figures/table can be reproduced. The artifact contains instructions to run the full 24h campaign for the interested readers.

More complete and detailed instructions on how to test a library is available in the README file of the repository.

D.4 Major claims

- (C1) *Trade-off analysis & Comparison with state-of-the-art*: libErator exposes the trade-off between creating and testing driver. This is showcased in experiment E1 which runs libErator with four different balance of testing and driver creation duration.

  The trade-off results are presented in Figure 3 while the comparison with the state-of-the-art is in Table 4, 5, & 6.

  Due to time constraints, we do not provide automatic scripts to run and extract the data from our competitors (i.e., Hopper, UTopia, FuzzGen, OSS-Fuzz, and OSS-Fuzz-Gen) that would be necessary for Table 4, Table 5, & Table 6. We, however, provide instructions on how to run these experiments.

- (C2) *Library exploration capability*: By diversifying the driver set, libErator explores different regions of a library. Our experiment E2 highlights on the 15 libraries we tested. In

the paper, the results are available in Figure 4 & Figure 5. The plot can slightly differ from the ones in the paper due to the different hardware configurations, but we expect the global trend across the benchmark to remain consistent.

- (C3) *Ablation study*: To highlight the contributions of each module of LIBERATOR, we conduct an ablation study to assess the cost and the gained coverage due to each. The results are presented in Table 9 & 10 and can be reproduced through the experiment E3.

D.5 Evaluation

In this section, we provide the detailed steps to run the experiments and process the results to get the tables presented in the paper. Overall, this process requires around two days of computation time on a server with 32 cores. Differently from the original paper, we provide a short version of the experiments that last 10 hours and require around 500GB of free space. We provide the documentation to reproduce the whole experiments in the repository.

We recommend running these experiments in `tmux` or `screen` session to avoid interruptions.

Experiment 1 (E1) - Claim (C1): Trade-off analysis:

[2 humans minutes + 60 compute-hours] The experiment leverages LIBERATOR to explore how each target library behave along the trade-off between t_{gen} and t_{test}. The experiment consists of running four campaigns with each a different ratio of t_{gen} to t_{test} and plotting the results.

[Preparation] Ensure that to have the results from the static analysis which are provided in the repository.

```
./install_analysis_result.sh
# Otherwise the static analysis will be run as part of the other commands but can
  ↪  take more than a day to complete.

./run_campaign_artifact.sh
# Expected output: You should see multiple docker builds and have logs for the
  ↪  different driver created and run.
```

[Execution] Run the commands:

```
# Kick the the campaigns necessary. We proceed firs to t_gen of 24h and reuse
  ↪  partial results for smaller t_gen values.
./fig3.sh
./tab4.py
```

[Results] Upon completion, the script will generate a PNG image identical to Figure 3 as `fig3.pdf`. Additionally, it will output a table similar to the first five columns of Table 4. There might be slight variations due to the different underlying machine and the inherent stochastic nature of fuzzing.

Experiment 2 (E2) - Claim (C2): Library exploration capability: [2 humans minutes + 10 compute-hours]

This experiment evaluates the capability of LIBERATOR to explore a library by computing the code and API function coverage across a 24h generation session. It reuses the results from experiment E1.

[Preparation] Make sure to have run the preparation for experiment E1.

[Execution] In a shell, run the following command:

```
# Compute the cumulative coverage across the drivers generated as part of E1.
# Expected output: The plots are stored in comulative_coverage and api_coverage
  ↪ folder, respectively.
./fig4.sh
./fig5.sh
```

[Results] Upon completion, the script will generate figures similar to the ones in Figure 4 & 5. There might be slight variations due to the different underlying machine and the inherent stochastic nature of fuzzing.

Experiment 3 (E3) - Claim (C3): Ablation study: [2 humans minutes + 24 compute-hours] This experiment evaluates the contribution of each module of LIBERATOR by conducting an ablation study.

[Preparation] Make sure to have run the preparation of experiment E1 first to have the numbers for *full* LIBERATOR in Table 9.

[Execution] In a shell, run the following command:

```
# Run libErator without field bias.
# Expected output: Tables similar to Table 9 and 10 in the paper will be printed.
./tab9.sh
./tab10.sh
```

[Results] Upon completion, the script will print tables similar to Table 9 & 10. There might be slight variations due to the different underlying machine and the inherent stochastic nature of fuzzing.

Appendix E

Ghostwriting acknowledgments

Figure E.1 – Thanks for your help, Coline!

Bibliography

[1] Marc Andreessen. «Why Software Is Eating The World». In: *The Wall Street Journal* 8 (2011), p. 20.

[2] Apache Software Foundation. *Xalan-C++ version 1.10*. https://xml.apache.org/xalan-c/.

[3] Apple Developers. *MALLOC_SIZE*. https://developer.apple.com/library/archive/documentation/System/Conceptual/ManPages_iPhoneOS/man3/malloc_size.3.html. 2006.

[4] Bence Babati and Norbert Pataki. «Comprehensive performance analysis of C++ smart pointers». In: *Pollack Periodica* 12.3 (2017), pp. 157–166.

[5] Domagoj Babic, Stefan Bucur, Yaohui Chen, Franjo Ivancic, Tim King, Markus Kusano, Caroline Lemieux, László Szekeres, and Wei Wang. «FUDGE: Fuzz Driver Generation at Scale». In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019.

[6] Nicolas Badoux, Flavio Toffalini, Yuseok Jeon, and Mathias Payer. «type++: Prohibiting Type Confusion With Inline Type Information». In: (Feb. 2025). DOI: 10.14722/ndss.2025.23053. URL: https://dx.doi.org/10.14722/ndss.2025.23053.

[7] Nicolas Badoux, Flavio Toffalini, and Mathias Payer. «Sourcerer: channeling the void». In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2025.

[8] Saam Barati and M Saboff. *Introducing the Jetstream 2 benchmark suite*. 2019. URL: https://webkit.org/blog/8685/introducing-the-jetstream-2-benchmark-suite.

[9] Daniel J Bernstein and Frank Denis. *libsodium secure memory*. https://libsodium.gitbook.io/doc/memory_management. 2014.

[10] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. «Translating Code Comments to Procedure Specifications». In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: Association for Computing Machinery, 2018, pp. 242–253. ISBN: 9781450356992. DOI: 10.1145/3213846.3213872. URL: https://doi.org/10.1145/3213846.3213872.

[11] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. «Directed Greybox Fuzzing». In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 2329–2344. ISBN: 9781450349468. DOI: 10.1145/3133956.3134020. URL: https://doi.org/10.1145/3133956.3134020.

[12] Gary Bradski et al. «OpenCV». In: *Dr. Dobb's journal of software tools* (2000).

[13] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. «SPEC CPU2017: Next-generation compute benchmark». In: *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 2018, pp. 41–42.

[14] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. «Control-flow integrity: Precision, security, and performance». In: *ACM Computing Surveys (CSUR)* 50.1 (2017), pp. 1–33.

[15] Marcel Busch, Aravind Machiry, Chad Spensky, Giovanni Vigna, Christopher Kruegel, and Mathias Payer. «TEEzz: Fuzzing trusted applications on cots android devices». In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2023, pp. 1204–1219.

[16] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. «Hawkeye: Towards a desired directed grey-box fuzzer». In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 2095–2108.

[17] Peng Chen and Hao Chen. «Angora: Efficient fuzzing by principled search». In: *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2018, pp. 711–725.

[18] Peng Chen, Yuxuan Xie, Yunlong Lyu, Yuxiao Wang, and Hao Chen. «HOPPER: Interpretative fuzzing for libraries». In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 1600–1614.

[19] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. «SAVIOR: Towards bug-driven hybrid testing». In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1580–1596.

[20] Chromium Developers. *Control Flow Integrity - The Chromium Projects*. URL: https://www.chromium.org/developers/testing/control-flow-integrity/.

[21] Clang Developers. *Clang 19 documentation*. https://clang.llvm.org/docs/DiagnosticsReference.html#wuninitialized.

[22] Clang Developers. *UBSan*. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#available-checks. 2016.

[23] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Michael Frantzen, and Jamie Lokier. «FormatGuard: Automatic Protection From printf Format String Vulnerabilities.» In: *USENIX Security Symposium*. Vol. 91. Washington, DC. 2001.

[24] cppreference.com. *C++ Union*. `https://en.cppreference.com/w/cpp/language/union`.

[25] Sayonara De Zoysa. *Microsoft global outages caused by CrowdStrike software glitch*. 2024.

[26] Defense Advanced Research Projects Agency (DARPA). *TRACTOR: Translating All C to Rust*. `https://www.darpa.mil/research/programs/translating-all-c-to-rust`.

[27] Christian DeLozier, Richard Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo MK Martin, and Steve Zdancewic. «Ironclad C++ a library-augmented type-safe subset of C++». In: *ACM SIGPLAN Notices* 48.10 (2013), pp. 287–304.

[28] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. «Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models». In: *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 2023, pp. 423–435.

[29] Vikram Dhillon, David Metcalf, Max Hooper, Vikram Dhillon, David Metcalf, and Max Hooper. «The DAO hacked». In: *blockchain enabled applications: Understand the blockchain Ecosystem and How to Make it work for you* (2017), pp. 67–78.

[30] Gregory J. Duck and Roland HC Yap. «EffectiveSan: type and memory error detection using dynamically typed C/C++». In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 2018.

[31] Xiaokang Fan, Sifan Long, Chun Huang, Canqun Yang, and Fa Li. «Accelerating Type Confusion Detection by Identifying Harmless Type Castings». In: *Proceedings of the 20th ACM International Conference on Computing Frontiers*. 2023, pp. 91–100. ISBN: 9798400701405.

[32] Sebastian Fernandez and Microsoft Security Response Team. *A proactive approach to more secure code*. `https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/`. July 2019.

[33] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. «AFL++ : Combining Incremental Steps of Fuzzing Research». In: *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. URL: `https://www.usenix.org/conference/woot20/presentation/fioraldi`.

[34] Gordon Fraser and Andrea Arcuri. «Evosuite: automatic test suite generation for object-oriented software». In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, pp. 416–419.

[35] Gordon Fraser and Andrea Arcuri. «EvoSuite: automatic test suite generation for object-oriented software». In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, pp. 416–419.

[36] Yasuhiro Fujiwara, Go Irie, and Tomoe Kitahara. «Fast algorithm for affinity propagation». In: *Twenty-Second International Joint Conference on Artificial Intelligence*. 2011.

[37] Shuitao Gan, Chao Zhang, Peng Chen, Bodong Zhao, Xiaojun Qin, Dong Wu, and Zuoning Chen. «GREYONE: Data Flow Sensitive Fuzzing». In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2577–2594. ISBN: 978-1-939133-17-5. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/gan.

[38] Alex Gaynor. *What science can tell us about C and C++'s security*. https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/. 2020.

[39] Michael Gibbs and Bjarne Stroustrup. «Fast dynamic casting». In: *Software: Practice and Experience* 36.2 (2006), pp. 139–156.

[40] GitHub staff. *Octoverse: AI leads Python to top language as the number of global developers surges*. https://github.blog/news-insights/octoverse/octoverse-2024/#the-most-popular-programming-languages. 2024.

[41] Patrice Godefroid. *Fuzzing: Using automated testing to identify security bugs in software*. 2020. URL: https://www.microsoft.com/en-us/research/blog/a-brief-introduction-to-fuzzing-and-why-its-an-important-tool-for-developers/.

[42] Patrice Godefroid, Michael Y Levin, and David Molnar. «SAGE: Whitebox Fuzzing for Security Testing: SAGE has had a remarkable impact at Microsoft.» In: *Queue* 10.1 (2012), pp. 20–27.

[43] Google. *Protocol Buffers*. URL: https://developers.google.com/protocol-buffers/.

[44] Google. *Structure-Aware Fuzzing with libFuzzer*. https://github.com/google/fuzzing/blob/bb05211c12328cb16327bb0d58c0c67a9a44576f/docs/structure-aware-fuzzing.md. 2020.

[45] Google. *What is V8?* https://v8.dev/.

[46] On2 Technologies / Google. *libvpx*. https://chromium.googleource.com/webm/libvpx. 2023.

[47] Harrison Green and Thanassis Avgerinos. «GraphFuzz: library API fuzzing with lifetime-aware dataflow graphs». In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1070–1081. ISBN: 9781450392211. DOI: 10.1145/3510003.3510228. URL: https://doi.org/10.1145/3510003.3510228.

[48] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. «FUZZILLI: Fuzzing for JavaScript JIT Compiler Vulnerabilities». In: *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023. URL: https://www.ndss-symposium.org/ndss-paper/fuzzilli-fuzzing-for-javascript-jit-compiler-vulnerabilities/.

[49] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. «TypeSan: Practical type confusion detection». In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, pp. 517–528.

[50] John L. Henning. «SPEC CPU2006 benchmark descriptions». In: *ACM SIGARCH Computer Architecture News* 34.4 (2006), pp. 1–17.

[51] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. «Interprocedural pointer alias analysis». In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21.4 (1999), pp. 848–894.

[52] Junwha Hong, Wonil Jang, Mijung Kim, Lei Yu, Yonghwi Kwon, and Yuseok Jeon. «CMASan: Custom Memory Allocator-aware Address Sanitizer». In: *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society. 2024, pp. 74–74.

[53] Google inc. *A Framework for Fuzz Target Generation and Evaluation*. `https://github.com/google/oss-fuzz-gen`. 2023.

[54] Google inc. *Fuzz target generation using LLMs*. `https://google.github.io/oss-fuzz/research/llms/target_generation/`. 2023.

[55] ISO C++ Standards Committee and others. *Standard for Programming Language C++. Working Draft N4950*. Tech. rep. Tech. rep. ISO IEC JTC1/SC22, 2023.

[56] Kyriakos Ispoglou, Daniel Austin, Vishwath Mohan, and Mathias Payer. «FuzzGen: Automatic Fuzzer Generation». In: *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2271–2287. ISBN: 978-1-939133-17-5. URL: `https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou`.

[57] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. «Block Oriented Programming: Automating Data-Only Attacks». In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: Association for Computing Machinery, 2018, pp. 1868–1882. ISBN: 9781450356930. DOI: 10.1145/3243734.3243739. URL: `https://doi.org/10.1145/3243734.3243739`.

[58] Jaakko Järvi, Jeremiah Willcock, and Andrew Lumsdaine. «Concept-controlled polymorphism». In: *Generative Programming and Component Engineering: Second International Conference, GPCE 2003, Erfurt, Germany, September 22-25, 2003. Proceedings 2*. Springer. 2003, pp. 228–244.

[59] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. «HexType: Efficient Detection of Type Confusion Errors for C++». In: *CCS*. 2017.

[60] Bokdeuk Jeong, Joonun Jang, Hayoon Yi, Jiin Moon, Junsik Kim, Intae Jeon, Taesoo Kim, WooChul Shim, and Yong Ho Hwang. «UTopia: Automatic Generation of Fuzz Driver using Unit Tests». In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 2676–2692. DOI: 10.1109/SP46215.2023.10179394.

[61] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. «Cyclone: a safe dialect of C». In: *USENIX Annual Technical Conference, General Track*. 2002, pp. 275–288.

[62] Phillip Johnston and Rozi Harris. «The Boeing 737 MAX saga: lessons for software organizations». In: *Software Quality Professional* 21.3 (2019), pp. 4–12.

[63] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. «WINNIE: Fuzzing windows applications with harness synthesis and fast cloning». In: *Proceedings of the 2021 Network and Distributed System Security Symposium (NDSS 2021)*. 2021.

[64] Paul Kehrer. *Memory unsafety in Apple's operating systems*. `https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/`. 2019.

[65] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[66] Jakob Koschel, Pietro Borrello, Daniele Cono D'Elia, Herbert Bos, and Cristiano Giuffrida. «UNCONTAINED: Uncovering Container Confusion in the Linux Kernel». In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 5055–5072. ISBN: 978-1-939133-37-3. URL: `https://www.usenix.org/conference/usenixsecurity23/presentation/koschel`.

[67] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. «Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security». In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 2013, pp. 721–732.

[68] Chris Lattner and Vikram Adve. «LLVM: A compilation framework for lifelong program analysis & transformation». In: *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE. 2004, pp. 75–86.

[69] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. «Preventing Use-after-free with Dangling Pointers Nullification». In: *22th Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8 - 11, 2015*. 2015.

[70] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. «Type casting verification: Stopping an emerging attack vector». In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 81–96.

[71] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. «Type casting verification: Stopping an emerging attack vector». In: *24th USENIX Security Symposium (USENIX Security 15)*. 2015, pp. 81–96.

[72] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. «CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models». In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2023, pp. 919–931. DOI: 10.1109/ICSE48619.2023.00085.

[73] Nancy G Leveson and Clark S Turner. «An investigation of the Therac-25 accidents». In: *Computer* 26.7 (1993), pp. 18–41.

[74] Qiang Liu, Flavio Toffalini, Yajin Zhou, and Mathias Payer. «VIDEZZO: Dependency-aware Virtual Device Fuzzing». In: *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society. 2023, pp. 3228–3245.

[75] Yuwei Liu, Yanhao Wang, Xiangkun Jia, Zheng Zhang, and Purui Su. «AFGEN: Whole-Function Fuzzing for Applications and Libraries». In: *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, 2024, pp. 11–11. DOI: 10.1109/SP54263.2024.00011. URL: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00011.

[76] LLVM. *LLVM Control Flow Integrity: Shared library support*. https://clang.llvm.org/docs/ControlFlowIntegrity.html#cfi-cross-dso.

[77] LLVM Developers. *LLVM-CFI: cast checking strictness*. https://clang.llvm.org/docs/ControlFlowIntegrity.html#cfi-strictness.

[78] LLVM Developers. *Smaller debug info with constructor type homing*. https://blog.llvm.org/posts/2021-04-05-constructor-homing-for-debug-info/.

[79] LLVM Developers. *Type Metadata*. llvm.org/docs/TypeMetadata.html.

[80] LLVM Developers. *TySan: A type sanitizer*. https://lists.llvm.org/pipermail/llvm-dev/2017-April/111766.html.

[81] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas Reps. «Debugging via run-time type checking». In: *International Conference on Fundamental Approaches to Software Engineering*. Springer. 2001, pp. 217–232.

[82] Stephan Lukasczyk and Gordon Fraser. «Pynguin: Automated Unit Test Generation for Python». In: *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 168–172. ISBN: 9781450392235. DOI: 10.1145/3510454.3516829. URL: https://doi.org/10.1145/3510454.3516829.

[83] Sadie Macintyre-Randall. «Enforcing C++ type integrity with fast dynamic casting, member function protections and an exploration of C++ beneath the surface». PhD thesis. University of Kent, 2023.

[84] Philipp Mao, Elias Valentin Boschung, Marcel Busch, and Mathias Payer. «Exploiting Android's Hardened Memory Allocator». In: *Proceeding of the 18th USENIX WOOT Conference on Offensive Technologies*. 2024.

[85] Nicholas D. Matsakis and Felix S. Klock. «The rust language». In: *ACM SIGAda Ada Letters* 34.3 (2014), pp. 103–104.

[86] Tom Mens and Tom Tourwé. «A survey of software refactoring». In: *IEEE Transactions on software engineering* 30.2 (2004), pp. 126–139.

[87] Microsoft. *Project Verona*. `https://github.com/microsoft/verona`.

[88] Shabnam Mirshokraie, Ali Mesbah, and Karthik Pattabiraman. «JSEFT: Automated javascript unit test generation». In: *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE. 2015, pp. 1–10.

[89] Paul Muntean, Sebastian Wuerl, Jens Grossklags, and Claudia Eckert. «CASTSAN: Efficient detection of polymorphic C++ object type confusions with LLVM». In: *Computer Security: 23rd European Symposium on Research in Computer Security, ESORICS 2018, Barcelona, Spain, September 3-7, 2018, Proceedings, Part I 23*. Springer. 2018, pp. 3–25.

[90] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. «CETS: compiler enforced temporal safety for C». In: *Proceedings of the 2010 international symposium on Memory management*. 2010, pp. 31–40.

[91] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. «SoftBound: Highly compatible and complete spatial memory safety for C». In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2009, pp. 245–258.

[92] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. «CCured: Type-safe retrofitting of legacy software». In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 27.3 (2005), pp. 477–526.

[93] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. «ParmeSan: Sanitizer-guided Greybox Fuzzing». In: *29th USENIX Security Symposium (USENIX Security 20)*. 2020, pp. 2289–2306.

[94] Carlos Pacheco and Michael D Ernst. «Randoop: feedback-directed random testing for Java». In: *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 2007, pp. 815–816.

[95] Chengbin Pang, Yunlan Du, Bing Mao, and Shanqing Guo. «Mapping to bits: Efficiently detecting type confusion errors». In: *Proceedings of the 34th Annual Computer Security Applications Conference*. 2018, pp. 518–528.

[96] Nikolaos S Papaspyrou. «A formal semantics for the C programming language». In: *Doctoral Disseration. National Technical University of Athens. Athens (Greece)* 15 (1998), p. 19.

[97] Mathias Payer. *Type Confusion: Discovery, Abuse, Protection*. `https://hexhive.epfl.ch/publications/files/18SyScan360-presentation.pdf`. Singapore, 2017.

[98] Tomas Plachetka. «POV-Ray: persistence of vision parallel raytracer». In: *Proc. of Spring Conf. on Computer Graphics, Budmerice, Slovakia*. Vol. 123. 1998, p. 129.

[99] The Chromium Project. *Memory safety*. https://www.chromium.org/Home/chromium-security/memory-safety/. 2025.

[100] Alex Rebert and Christoph Kern. *Secure by Design: Google's Perspective on Memory Safety*. Tech. rep. Google Security Engineering, 2024.

[101] Greg Roelofs. *libpng*. http://www.libpng.org/pub/png/libpng.html. 2023.

[102] Herbert Schildt. «C the complete reference». In: (2021).

[103] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. «kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels». In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 167–182. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo.

[104] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. «Counterfeit Object-oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications». In: *2015 IEEE Symposium on Security and Privacy*. 2015, pp. 745–762. DOI: 10.1109/SP.2015.51.

[105] Ivannikov Institute for System Programming of the Russian Academy of Sciences. *CASR: Crash Analysis and Severity Report*. https://github.com/ispras/casr. 2023.

[106] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. «AddressSanitizer: A fast address sanity checker». In: *2012 USENIX annual technical conference (USENIX ATC 12)*. Boston, MA: USENIX Association, June 2012, pp. 309–318. ISBN: 978-931971-93-5. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[107] Konstantin Serebryany and Timur Iskhodzhanov. «ThreadSanitizer: data race detection in practice». In: *Proceedings of the workshop on binary instrumentation and applications*. 2009, pp. 62–71.

[108] Kosta Serebryany. «Continuous fuzzing with libfuzzer and AddressSanitizer». In: *2016 IEEE Cybersecurity Development (SecDev)*. IEEE. 2016, pp. 157–157.

[109] Kostya Serebryany. «OSS-Fuzz - Google's continuous fuzzing service for open source software». In: Vancouver, BC: USENIX Association, Aug. 2017.

[110] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. «On the effectiveness of address-space randomization». In: *Proceedings of the 11th ACM conference on Computer and communications security*. 2004, pp. 298–307.

[111] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. «SoK: Sanitizing for Security». In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 1275–1295. DOI: 10.1109/SP.2019.00010.

[112] Evgeniy Stepanov and Konstantin Serebryany. «MemorySanitizer: fast detector of uninitialized memory use in C++». In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2015, pp. 46–55.

[113] Milan Stevanovic. *Advanced C and C++ Compiling*. 1st. USA: Apress, 2014. ISBN: 1430266678.

[114] Bjarne Stroustrup. *How do I define an in-class constant*. URL: https://www.stroustrup.com/bs_faq2.html\#in-class.

[115] Yulei Sui and Jingling Xue. «SVF: interprocedural static value-flow analysis in LLVM». In: *Proceedings of the 25th international conference on compiler construction*. 2016, pp. 265–266.

[116] Herb Sutter. *C++ safety, in context*. 2024. URL: https://herbsutter.com/2024/03/11/safety-in-context/.

[117] Herb Sutter, Bjarne Stroustrup, and other contributors. *C++ Core Guidelines*. 2015. URL: https://github.com/isocpp/CppCoreGuidelines/commit/6156e957827599f2fcaa5401ebb1668ae9edcdc8.

[118] LLVM Team. *LLVM profdata merge*. https://llvm.org/docs/CommandGuide/llvm-profdata.html#profdata-merge. 2013.

[119] The Clang Team. *SanitizerCoverage*. https://clang.llvm.org/docs/SanitizerCoverage.html. 2023.

[120] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. «Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM». In: *23rd USENIX Security Symposium*. 2014.

[121] Flavio Toffalini, Nicolas Badoux, Zurab Tsidnadze, and Mathias Payer. *Artifact for* LIBERATOR. 2025. DOI: 10.5281/zenodo.15201791. URL: https://doi.org/10.5281/zenodo.15201791.

[122] Flavio Toffalini, Nicolas Badoux, Tsinadze Zurab, and Mathias Payer. «Liberating Libraries through Automated Fuzz Driver Generation». In: *Proceedings of the ACM on Software Engineering,* ACM New York, NY, USA, July 2025. DOI: 10.1145/3729365. URL: https://dx.doi.org/10.14722/3729365.

[123] Victor Van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. «Memory errors: The past, the present, and the future». In: *Research in Attacks, Intrusions, and Defenses: 15th International Symposium, RAID 2012, Amsterdam, The Netherlands, September 12-14, 2012. Proceedings 15*. Springer. 2012, pp. 86–106.

[124] Arjan van de Ven. *Exec shield*. https://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf. 2004.

[125] Mattia Vivanti, Andre Mis, Alessandra Gorla, and Gordon Fraser. «Search-based data-flow test generation». In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE. 2013, pp. 370–379.

[126] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. «High system-code security with low overhead». In: *2015 IEEE Symposium on Security and Privacy*. IEEE. 2015, pp. 866–879.

[127] Xi Wang, Nickolai Zeldovich, M Frans Kaashoek, and Armando Solar-Lezama. «Towards optimization-safe systems: Analyzing the impact of undefined behavior». In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 2013, pp. 260–275.

[128] Yanhao Wang, Xiangkun Jia, Yuwei Liu, Kyle Zeng, Tiffany Bao, Dinghao Wu, and Purui Su. «Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization.» In: *NDSS*. 2020.

[129] Windows Developers. *_msize*. `https://learn.microsoft.com/en-us/cpp/c-runtime-library/reference/msize?view=msvc-170`. 2023.

[130] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. «Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities». In: *Proceedings of the 40th International Conference on Software Engineering*. 2018, pp. 327–337.

[131] Li Yujian and Liu Bo. «A normalized Levenshtein distance metric». In: *IEEE transactions on pattern analysis and machine intelligence* 29.6 (2007), pp. 1091–1095.

[132] Lyu Yunlong, Xie Yuxuan Chen Peng, and Chen Hao. «Prompt Fuzzing for Fuzz Driver Generation». In: *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security*. CCS '24. Association for Computing Machinery, 2024.

[133] Michal Zalewski. *american fuzzy lop*. `https://lcamtuf.coredump.cx/afl/`. 2013.

[134] Hamzeh Zawawy and Jon Bottarini. *Android goes all-in on fuzzing*. 2023. URL: `https://security.googleblog.com/2023/08/android-goes-all-in-on-fuzzing.html`.

[135] Yizhuo Zhai, Zhiyun Qian, Chengyu Song, Manu Sridharan, Trent Jaeger, Paul Yu, and Srikanth V Krishnamurthy. «Don't Waste My Efforts: Pruning Redundant Sanitizer Checks Type Checks». In: *33rd USENIX Security Symposium*. 2024.

[136] Cen Zhang, Yuekang Li, Hao Zhou, Xiaohan Zhang, Yaowen Zheng, Xian Zhan, Xiaofei Xie, Xiapu Luo, Xinghua Li, Yang Liu, et al. «Automata-Guided Control-Flow-Sensitive Fuzz Driver Generation». In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 2867–2884. ISBN: 978-1-939133-37-3. URL: `https://www.usenix.org/conference/usenixsecurity23/presentation/zhang-cen`.

[137] Cen Zhang, Xingwei Lin, Yuekang Li, Yinxing Xue, Jundong Xie, Hongxu Chen, Xinlei Ying, Jiashui Wang, and Yang Liu. «APICraft: Fuzz Driver Generation for Closed-source SDK Libraries». In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 2811–2828.

[138] Mingrui Zhang, Jianzhong Liu, Fuchen Ma, Huafeng Zhang, and Yu Jiang. «IntelliGen: Automatic driver synthesis for fuzz testing». In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2021, pp. 318–327.

[139] Mingrui Zhang, Chijin Zhou, Jianzhong Liu, Mingzhe Wang, Jie Liang, Juan Zhu, and Yu Jiang. «DAISY: Effective Fuzz Driver Synthesis with Object Usage Sequence Analysis». In: *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 2023, pp. 87–98. DOI: 10.1109/ICSE-SEIP58684.2023.00013.

[140] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. «FISHFUZZ: catch deeper bugs by throwing larger nets». In: *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association. 2023, pp. 1343–1360.

Rue de la Gare 21
1030 Bussigny
CH—Switzerland
Swiss citizen—married
Born 06.11.1994

Nicolas Badoux

n.badoux@hotmail.com +41 79 914 00 47 nbadoux

Education

Doctor of Sciences (PhD) *2020–2025*
École Polytechnique Fédérale de Lausanne (EPFL) - Switzerland
- Advisor: Prof. Mathias Payer in the HexHive laboratory.
- Thesis: Securing low-level code with minimal developer efforts.
- Topics: System security, software testing, compiler-based defenses, fuzzing.

Master of Science ETH in Computer Science *2016–2019*
Eidgenössische Technische Hochschule Zürich (ETHZ) - Switzerland
- Specialization in Information Security, GPA: 5.39/6.

Bachelor in Communication Sciences *2013–2016*
École Polytechnique Fédérale de Lausanne (EPFL) - Switzerland
- Exchange program @ **Carnegie Mellon University** - USA, GPA: 5.26/6. *2015–2016*

Billingual Matura (German/French) *2010–2013*
Kantonschule Frauenfeld & Gymnase d'Yverdon - Switzerland
- Specialization in Mathematics and Physics, GPA: 5.19/6, Best 3%.

Research Experience

`type++`: prohibiting type confusion with inline type information *NDSS'25*
Authors: **Nicolas Badoux**, Flavio Toffalini, Yuseok Jeon, & Mathias Payer.
- *Distinguished Paper Award* (top 5%).
- In C++, incorrect downcast are a severe vulnerability often exploited in the wild.
- By inlining the type in each C++ object, we create a compiler-based mitigation against type confusion attacks allowing downcast to be checked at runtime while requiring minimal code adaptations. We evaluate our prototype against the state-of-the-art and achieve less than 1% runtime overhead while protecting 90B casts. We deploy our prototype on Chromium.
- Built on top of LLVM, `type++` is available on GitHub and its artifact evaluated.
- During this multi-year project, I learned some intricacies of compilers, developed my writing skills, and strategic planing to face a constantly evolving project.

libErator: Balancing library fuzzing without consumer code *FSE'25*
Authors: Flavio Toffalini, **Nicolas Badoux**, Zurab Tsinadze, & Mathias Payer.
- Drivers, a sequence of API calls building state, allows for dynamic testing like fuzzing, to execute a library's code. Manually written drivers are rare and exhaustively tested.
- libErator automates the generation of fuzzing drivers without consumer code and allow for balancing resources between driver generation and fuzzing.
- From insights gathered through LLVM passes, we build valid C drivers calling the API.
- We report and fix 24 bugs, including CVE-2024-8006. We release our prototype on Github.
- Through the design and multifaceted evaluation of libErator, I improved my cross-cutting understanding of complex systems.

Sourcerer: channeling the void *DIMVA'25*
Authors: **Nicolas Badoux**, Flavio Toffalini, & Mathias Payer.
- In C++, conversions from `void*` to typed pointers are ubiquitous but, if the type is not the original one, lead to type confusions and possibly further memory corruption.
- By extending the protection of type++ to all the types used in casts, we design Sourcerer, a complete type confusions sanitizer. With a low-overhead of 5% on average, we conduct the first fuzzing campaign targeting specifically type confusions.
- We find type confusions in Blender and OpenCV and release our prototype on GitHub.
- As the main author, I designed and evaluated our system as well as wrote the paper.

Bypassing LLVM-CFI cast protection *Ongoing*

Authors: Nicolas Almerge, **Nicolas Badoux**, & Mathias Payer.

- We present a novel attack against LLVM-CFI, bypassing the cast protection for C++.
- As the main advisor for this Master project, I laid out the research plan, provided guidance, and reviewed the results.

Industry Experience

Software Engineer - Fondation Digger, NGO - Tavannes, CH *Aug' 2019–March 2020*

- Developed a virtual overlay for remotely removing explosives with the help of OpenCV and Unity in an Agile environment as part of my civil service.

Software Engineer - Compassion Suisse, NGO - Yverdon, CH *March–May 2018*

- As part of my civil Service, contributed to open source Python modules for the Odoo ERP.

Security Engineer Intern - Ergon Informatik - Zürich, CH *60%—Sept' 2017–March 2018*

- Developed a blackbox fuzzer in Python to find bugs in Ergon's Web Application Firewall.

Technology Summer Analyst - Morgan Stanley - London, UK *June–Aug' 2016*

- Developed charts in AngularJS for statistics of the Architecture Security team.

Skills

Programming Languages: Python, C++, LaTeX, Bash.

Software: LLVM, Docker, GDB, Linux, libfuzzer.

Spoken Languages: French (native), English, Swiss-German, German.

Teaching Assistant

CS-119 Information, Calcul & Communication *2022 & 2024*

CS-323 Operating System *2021*

CS-412 Software Security *2021 & 2023*

COM-402 Information Security & Privacy *2023*

Activities

Board Member, Treasurer - Groupes Bibliques des Écoles et Universités *2023–ongoing*

- Define the vision, hiring of the general secretary, and budget planning ($\simeq$ 500kCHF).

Camp Leader - Interjeunes & Ligue pour la Lecture de la Bible *2014, 2017, 2021, 2022*

- Lead camps with up 110 kids/young adults for a week. Built a team, prepared the event, managed the team and was in charge of the authority during the week.

References

Prof. Dr. Mathias Payer *mathias.payer@nebelwelt.net*

- Associate Professor at EPFL in Lausanne (CH) and head of HexHive.
- Advised me during my PhD between 2020 and 2025.

Prof. Dr. Flavio Toffalini *flavio.toffalini@rub.de*

- Assistant Professor at Ruhr-Universität in Bochum (DE).
- Close collaborator and advising post-doc during my PhD (2021–2025).

Benoît Pfister *benoit.pfister@gbeu.ch*

- Chairman of the Board at Groupes Bibliques des Écoles et Universités.
- We worked together for hiring committees, budgeting, and general strategy.