PRACTICAL TYPE AND MEMORY SAFETY VIOLATION DETECTION MECHANISMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Yueseok Jeon

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2020

Purdue University

West Lafayette, Indiana

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF DISSERTATION APPROVAL

Dr. Mathias Payer, Co-Chair

Department of Computer Science

Dr. Changhee Jung, Co-Chair

Department of Computer Science

Dr. Byoungyoung Lee

Department of Computer Science

Dr. Xiangyu Zhang

Department of Computer Science

Dr. Pedro Fonseca

Department of Computer Science

Approved by:

Dr. Clifton W. Bingham

Head of the Department Graduate Program

Dedicated to my wife Hyunmin Lee, my son Taehyeon Jeon, and my family

ACKNOWLEDGMENTS

First and foremost, I would like to thank my major advisor Dr. Mathias Payer for giving me a precious chance to pursue my Ph.D. in the USA and for his invaluable guidance and support, as well as encouragement during my doctoral studies. I also would like to thank my co-advisors Dr. Byoungyoung Lee for his invaluable help and Dr. Changhee Jung for understanding and supporting me. To my committee members, Dr. Xiangyu Zhang and Dr. Pedro Fonseca, I would like to thank them for kindly agreeing to be on my committee, for their guidance, and for their valuable time. I would like to thank past and present colleagues in the HexHive group, Priyam Biswas, Hui Peng, Nathan Burow, Scott Carr, Prashast Srivastava, Derrick McKee, Bader AlBassam, Adrian Herrera, Kyriakos Ispoglou, Naif Almakhdhub, Abe Clements, Terry Hsu, Ahmed Hussein, Atri Bhattacharyya, Ahmad Hazimeh, Uros Tesic, Nicolas Badoux, Jelena Jankovic, Jean-Michel Crepel, Antony Vennard, and Andrés Sanchez, for their precious feedback, insightful discussions, and friendship. I also would like to acknowledge my friends and inspirations outside Purdue, Dr. Junghwan Rhee, Dr. Yonghwi Kwon, Dr. Chung Hwan Kim, Taegyu Kim, and Alexander Yurkov for their invaluable help and friendship. Last but not least, I would like to thank my family, especially my wife Hyunmin Lee, my son Taehyeon Jeon, my parents, and my in-laws, for their trust, sacrifice, and support on this long journey.

TABLE OF CONTENTS

Page

LIST OF TABLES . . . vii

LIST OF FIGURES . . . ix

ABBREVIATIONS . . . xi

ABSTRACT . . . xii

1 INTRODUCTION . . . 1

2 HEXTYPE: EFFICIENT DETECTION OF TYPE CONFUSION ERRORS FOR C++ . . . 7
- 2.1 HexType . . . 10
  - 2.1.1 Background . . . 10
  - 2.1.2 Threat Model . . . 17
  - 2.1.3 HexType Design and Implementation . . . 18
  - 2.1.4 Implementation . . . 31
  - 2.1.5 Evaluation . . . 32
  - 2.1.6 Discussion . . . 40
  - 2.1.7 Related Work . . . 40
- 2.2 Conclusion . . . 42

3 V-TYPE: INLINE TYPE INFORMATION TO COUNTER TYPE CONFUSION . . . 43
- 3.1 Background . . . 46
  - 3.1.1 Classes Hierarchies & Polymorphism . . . 46
  - 3.1.2 Type confusion . . . 48
  - 3.1.3 C++ Casting . . . 48
- 3.2 Design and Implementation . . . 50
  - 3.2.1 Vtable inline metadata structure . . . 51
  - 3.2.2 Addressing compatibility issues . . . 51
  - 3.2.3 Increase detection coverage . . . 55
  - 3.2.4 Only changing type related to type casting . . . 56
  - 3.2.5 Implementation . . . 57
- 3.3 Evaluation . . . 58
  - 3.3.1 Coverage on type casting . . . 58
  - 3.3.2 Performance Overhead . . . 59
- 3.4 Future Work . . . 60

Page

4 FUZZAN: EFFICIENT SANITIZER METADARTA DESIGN FOR FUZZING 61
4.1 Background and Analysis 64
4.1.1 Fuzzing overhead 65
4.1.2 Address Sanitizer 66
4.1.3 Overhead Analysis of Fuzzing with ASan 66
4.2 FuZZan design 68
4.2.1 FuZZan Metadata Structures 68
4.2.2 Dynamic metadata structure switching 73
4.3 Implementation 75
4.4 Evaluation 77
4.4.1 Detection capability 78
4.4.2 Efficiency of new metadata structures 79
4.4.3 Efficiency of dynamic metadata structure 83
4.4.4 Real-world fuzz testing 87
4.4.5 Bug finding effectiveness 87
4.4.6 FuZZan Flexibility 89
4.5 Discussion 91
4.6 Related Work 92
4.6.1 Reducing Fuzzing Overhead 92
4.6.2 Optimizing Sanitizers 93
4.7 Conclusion 95

5 THE DIRECTION OF FUTURE RESEARCH 96

6 SUMMARY 97

REFERENCES 98

VITA 107

LIST OF TABLES

Table Page

2.1 Detection coverage number 33

2.2 HashMap miss rate number 37

2.3 The number of safe objects identified by HexType's optimization algorithm. HexType does not keep track of these safe objects. 38

2.4 Performance Overhead 38

3.1 Detection coverage comparison with other pointer casting monitor approaches 55

3.2 The evaluation of typecasting verification coverage against SPEC CPU2006. The # indicates the number of verified casting operations during SPEC CPU2006 test. The k represents thousand, m represents million, and b represents billion. 58

3.3 SPEC CPU2006 benchmark performance overhead for Clang-CFI and V-Type. The first column with % denotes the ratio between Native and Clang-CFI. The next column denotes the ratio between Native and V-Type. 59

4.1 Comparison between native and ASan executions with a breakdown of time spent in memory management, and time spent for ASan's initialization and logging. Results are aggregated over 500,000 executions of the full Google fuzzer test suite [95]. Times are shown in milliseconds, and % denotes the ratio to total execution time. 64

4.2 Comparison of metadata structures. 69

4.3 Three different metadata structure modes' detection capability based on the Juliet Test Suite for memory corruption CWEs. FuZZan and ASan have identical results. Good tests have no memory corruption to check for false positives. Bad tests are intentionally buggy to check for false negatives. 78

4.4 Comparison between four min-shadow memory modes, RB tree, Native, and ASan execution overhead during input record and replay fuzz testing with empty and provided seed sets. The time (s) indicates the average of all 26 applications' execution time during testing. Positive percentage (e.g., 20%) denotes overhead while negative percentage indicates a speedup. 81

Table | Page

4.5 Comparison between FuZZan's three different optimization modes, native min-shadow memory (1G) mode, and min-shadow memory (1G) mode with FuZZan's two optimizations, and dynamic metadata structure switching (Dynamic) mode execution overhead during all 26 applications' input record and replay fuzz testing. 82

4.6 Comparison between native, ASan, min-shadow memory (1G), two optimizations with min-shadow memory executions with a breakdown of time spent in memory management, and time spent for ASan's initialization and logging. Results are aggregated over 500,000 executions of the full Google fuzzer test suite. Times are shown in milliseconds, and % denotes the ratio between single execution time and each section execution's time. 82

4.7 Evaluating FuZZan's total execution number and unique discovered path for 24 hours fuzz testing with provided seeds. The (M) denotes 1,000,000 (one million) and ratio (%) is the ratio between ASan and FuZZan. 86

4.8 Evaluating FuZZan's bug finding speed. The TTE denotes the mean time-to-exposure. The AF is assertion error and the BO denotes buffer overflow. 88

4.9 Comparison between Native, MSan, MSan-nolock, and min-shadow memory execution overhead during input record and replay fuzz testing with provided seed sets. MSan-nolock disables lock/unlock for MSan's logging depots. Time (s) indicates the average of execution time. Positive percentages denote overhead, negative percentages denote speedup. 89

LIST OF FIGURES

Figure Page

2.1 Visualization of an example C++ type hierarchy, showing the directions of (safe) upcasting and (unsafe) downcasting. 11

2.2 A code example and diagram of a type confusion problem where an ancestor class is incorrectly accessed using a pointer to a descendent class. The static cast results in type confusion and accessing the field D→y results in a memory safety violation. 12

2.3 A system overview of HexType. HexType consists of several modules that analyze type relationship information and insert object tracing and typecasting instrumentation to verify typecasting operation. 16

2.4 Code example for std::aligned_storage using placement new and `reinterpret_cast` to manage type allocation. 22

2.5 A snapshot example of object mapping table, showing how it maps an object using a combination of fast-path and slow-path slots. When HexType looks up type information, an object address is used to obtain the reference to the corresponding object mapping table entry. HexType first matches the fast-path slot. If not present in the fast-path slot, HexType then searches the corresponding red-black tree to find type information (slow-path) and updates the fast-path accordingly. 24

2.6 An example of how `reinterpret_cast` results in a type confusion problem.26

2.7 An example of how HexType creates a potentially unsafe object type set. In the example, we assume that objects of type B and C are typecast. HexType will identify these potentially unsafe types and all its children types as unsafe. 28

2.8 An example for safe and unsafe casting. The three examples (line 9, 12, and 16) are all safe casting. For the first two examples, each typecasting operation obtains the object address using the address-of operator and the array name. In the third example (line 14), we can simply determine the object type using use-def chain analysis. The last example (line 22) is an unsafe casting case when we cannot track the object type (i.e., external function). 30

Figure Page

2.9 Overview of HexType's implementation. HexType consists of several compiler passes in clang and LLVM that insert object tracing and typecasting instrumentation and a corresponding runtime library. 31

3.1 Code example for type confusion 47

3.2 Overview of V-Type's architecture and workflow. 50

3.3 Change non-polymorphic into polymorphic object. 51

3.4 Code example for adding placement_new before switching to union's a polymorphic object member. 53

3.5 Code example for defining the const variable of class outside the class . . . 54

3.6 An example of how V-Type creates a type set related to casting and only changes type using this set. In the example, we assume that the object of type B is typecast. V-Type will identify this type B and all its children types as casting related types. 56

4.1 Overview of FuZZan's architecture and workflow. 69

4.2 Design of FuZZan's customized RB-tree. 70

4.3 ASan and min-shadow memory modes' memory mapping on 64-bit platforms. ASan (top) reserves 20TB memory space for heap and shadow memory, conversely, min-shadow memory mode (bottom) reserves 4512MB memory space for heap and shadow memory. Each application's stack, heap, and other sections (BSS, data, and text) map to the corresponding shadow regions. Further, the shadow memory region is mapped inaccessible. 72

4.4 Evaluating the frequency of metadata structure switching and each metadata structure selection over the first 500,000 tests each for c-ares and vorbis in Google's fuzzer test suite and pngfix, size, and nm. The number on each bar indicates the total metadata switches. 84

ABBREVIATIONS

RTTI	Run-Time Type Information
CFI	Control Flow Integrity
TCB	Trusted Computing Base
ODR	One Definition Rule
CPI	Code Pointer Integrity
AFL	American Fuzzy Lop
ASAN	Address Sanitizer
KASAN	Kernel Address Sanitizer
MSAN	Memory Sanitizer
TSAN	Thread Sanitizer
OOM	Out Of Memory
RNG	Random Number Generator
SMT	Simultaneous MultiThreading

ABSTRACT

Yuseok Jeon Ph.D., Purdue University, December 2020. Practical type and memory safety violation detection mechanisms. Major Professor: Mathias Payer.

System programming languages such as C and C++ are designed to give the programmer full control over the underlying hardware. However, this freedom comes at the cost of type and memory safety violations which may allow an attacker to compromise applications.

In particular, type safety violation, also known as type confusion, is one of the major attack vectors to corrupt modern C++ applications. In the past years, several type confusion detectors have been proposed, but they are severely limited by high performance overhead, low detection coverage, and high false positive rates. To address these issues, we propose HexType and V-Type. First, we propose HexType, a tool that provides low-overhead disjoint metadata structures, compiler optimizations, and handles specific object allocation patterns. Thus, compared to prior work, HexType significantly improves detection coverage and reduces performance overhead. In addition, HexType discovers new type confusion bugs in real world programs such as Qt and Apache Xerces-C++. However, HexType still has considerable overhead from managing the disjoint metadata structure and tracking individual objects, and has false positives from imprecise object tracking, although HexType significantly reduces performance overhead and detection coverage. To address these issues, we propose a further advanced mechanism V-Type, which forcibly changes non-polymorphic types into polymorphic types to make sure all objects maintain type information. By doing this, V-Type removes the burden of tracking object allocation and deallocation and of managing a disjoint metadata structure, which reduces performance overhead and improves detection precision.

Another major attack vector is memory safety violations, which attackers can take advantage of by accessing out of bound or deleted memory. For memory safety violation detection, combining a fuzzer with sanitizers is a popular and effective approach. However, we find that heavy metadata structure of current sanitizers hinders fuzzing effectiveness. Thus, we introduce FuZZan to optimize sanitizer metadata structures for fuzzing. Consequently, FuZZan improves fuzzing throughput, and this helps the tester to discover more unique paths given the same amount of time and to find bugs faster.

In conclusion, my research aims to eliminate critical and common C/C++ memory and type safety violations through practical programming analysis techniques. For this goal, through these three projects, I contribute to our community to effectively detect type and memory safety violations.

1 INTRODUCTION

C/C++ are low-level programming languages used everywhere from most operating system kernels to modern browsers because of its high performance, rich function libraries, and low-level functionality, e.g., low-level memory access allowing the efficient handling of memory allocations and deallocations.

However, enforcing memory safety (i.e., only accessing their intended referents) and type safety (i.e., operations on the object always being compatible with the object's type) are left to the programmer. This lack of safety leads to memory and type safety violations that can cause programs to crash unexpectedly, silently to generate incorrect results, or to be abused to attack programs, allowing the attacker to gain full privileges of these programs.

Type safety violations occur when one data type is mistaken for another due to unsafe typecasting, leading to a reinterpretation of the underlying type representation in semantically mismatching contexts. For instance, a program may cast an instance of a parent class to a descendant class, even though this is neither safe nor allowed at the programming language level if the parent class lacks some of the fields or virtual functions of the descendant class. When the program subsequently uses the fields or functions, it may use data, say, as a regular field in one context and as a virtual table (Vtable) pointer in another. Such type safety violations are wide spread; many are found in a wide range of software products such as Google Chrome V8 (CVE-2020-6418), WebKitGTK (CVE-2020-3897 and CVE-2020-3901), Autodesk FBX SDK (CVE-2020-7081), Adobe Flash Player (CVE-2020-3757), PhantomPDF (CVE-2020-15638), Firefox (CVE-2019-17026), Libxslt (CVE-2019-5815), Ghostscript (CVE-2018-19134), Adobe Flash Playeri (CVE-2018-4944), and Foxit PDF Reader (CVE-2018-9942). These type safety violations are also security critical, as many are demonstrated to be easily exploitable due to deterministic runtime be-

haviors. At the same time, several type safety violation detection tools have also been proposed. These tools detect type confusion by using a pointer casting monitor [1–4] or a pointer usage monitor [5, 6]. However, these solutions are severely limited due to the high runtime performance overhead as well as low coverage for object tracing and detection.

Memory safety violations occur when an invalid pointer is dereferenced. More specifically, these violations can be classified into two types: (i) temporal safety violation, which occurs when accessing a referent that is no longer valid and (ii) spatial safety violation, which occurs when accessing the data at a location in memory that is outside the bounds of an allocated object. Address sanitizer (ASan), the most popular sanitizer, has detected over 10,000 memory safety bugs [7–9] in various applications (e.g., over 3,000 bugs in Chrome in 3 years [7]) and over 50 bugs [8] have been reported in the Linux Kernel. Additionally, memory safety violations are the most dangerous class of bugs, accounting for 70% of vulnerabilities at Microsoft [10].

To prevent the memory safety violations, various violation detection tools have been proposed for temporal safety violations [11–18] and spatial safety violations [11–16, 19–34]. Among them, the combination of a fuzzer with ASan is the most common approach to find memory safety violations, but some issues remain. Several of ASan's design choices (e.g., heavy metadata structure or unnecessary logging) are not geared towards a fuzz testing environment (i.e., highly repetitive and short execution) and reduce the benefit of combining the two.

Thus, to address these critical issues in type and memory safety detectors and to propose advanced detectors, my thesis statement is as follows:

> **Practical type and memory safety violation detection mechanisms, implemented through a combination of static and runtime compiler-based techniques, enable efficient detection of policy violations.**

In this thesis, we advance the state of the art for type and memory safety. More specifically, for enforcing type safety, we propose two practical type confusion detectors, which significantly improve detection coverage and remove performance overhead

to help find type confusion bugs quickly and accurately. As for memory safety, we identify and analyze the primary source of overhead that occurs when sanitizers are used with fuzzing, and in response we optimize the sanitizer for fuzzing to improve the throughput and detect memory safety violations faster. For our two practical type confusion detectors, first, we propose HexType, a mechanism that protects C++ software from type confusion by making all casts explicit. Each cast in the source language (explicit or implicit, static or dynamic) is turned into a dynamic runtime check. HexType records the type of each object and specific casts are replaced with our instrumentation. We fundamentally address the challenges of earlier work by (i) increasing coverage of typecasting checks and (ii) drastically reducing overhead. Our prototype implementation of HexType vastly outperforms state-of-the-art type confusion detectors, increasing coverage and often lowering overhead. Our reduced overhead is the result of novel optimization techniques and using an efficient type metadata structure. Due to our increased coverage, we discover four new type confusion vulnerabilities (which evaded previous approaches) in two widely-used open source libraries (Qt Base library and Apache Xerces-C++) during our evaluation. For the Firefox benchmarks, HexType increases coverage by 1.1 – 6.1 times compared to TypeSan with some increased performance overhead due to the vast increase in coverage. For SPEC CPU2006 benchmarks with overhead, we show a 2 – 33.4 times reduction in overhead.

Despite our advancements, disjoint metadata structure approaches including HexType incur considerable overhead because of the additional object tracking and metadata structure managing overhead. These approaches also have high false positive rate issues because of their incorrect object tracking. Thus, we propose V-Type to overcome these limitations. To improve precision and performance, we inline type information via Vtable pointers. However, as this information only exists in polymorphic objects, we must forcibly change non-polymorphic objects into polymorphic ones to ensure each object will maintain type information. When we forcibly insert type information into non-polymorphic objects, however, several challenges occur: (1)

we need to insert constructor calls into all object allocation sites through the `malloc` family to initialize a Vtable pointer, (2) we need to address standard violation issues when we forcibly insert a Vtable pointer, and (3) we need to address issues when the changed object will be used for communication with other components (e.g., libraries or kernel). To address these issues, V-Type inserts additional constructor calls to initialize a Vtable pointer of forcibly changed polymorphic objects. Additionally, V-Type addresses standard violation issues by modifying compilers without additional side effects. For communication issues with other components, V-Type rebuilds related components with itself to make sure all components have the same type layout. For optimization, V-Type only changes casting-related non-polymorphic objects into polymorphic objects. Since objects unrelated to casting will never be used for casting, V-Type does not need to change these non-polymorphic objects. To further increase detection coverage, V-Type also detects type confusion from void* or another unrelated type to the wrong dynamic type. According to our evaluation of SPEC CPU2006, compared to native execution, V-Type shows negligible overhead, only 1%, which is 11 times faster than HexType. Additionally, V-Type shows 10 times better detection coverage than Clang-CFI.

To detect memory safety violations, a combination of fuzzing and a sanitizer is the most widely used combination. More specifically, fuzzing is a powerful and widely used software security testing technique that randomly generates inputs to find bugs. Despite fuzzing's demonstrated bug-finding effectiveness, it only discovers relatively simple bugs without assistance, e.g., failed assertions or memory errors such as segmentation faults. Bugs that silently corrupt the program's memory state, but do not cause a crash, are missed. To detect such bugs, fuzzers must be paired with sanitizers that enforce additional security policies at runtime that make bugs detectable. Unfortunately, current fuzzer plus sanitizer combinations trade increased detection capabilities (more bug classes detected), for decreased throughput. The decreased throughput is a direct result of the substantial overhead, e.g., up to 6.59× for ASan, imposed by sanitizers on instrumented software. Thus, we identify and analyze the

primary source of overhead when ASan is used with fuzzing, and pinpointing the design decisions that cause the overhead. We design and implement a sanitizer optimization (FuZZan) and apply it to ASan; that is, we design several new metadata structures along with a dynamic metadata structure switching to choose the optimal structure at runtime. Consequently, FuZZan improves performance over ASan by 48% with Google's seed corpus, while detecting the same classes of vulnerabilities.

Contributions

We present novel mechanisms for detecting type and memory safety violations. The work presented in this dissertation has all been peer reviewed and published, with the exception of the V-Type project which is still in preparation for submission. In particular, our advanced type confusion detector, HexType: Efficient Detection of Type Confusion Errors for C++ [35] appeared at CCS '17, and our optimized sanitizers for fuzzing, FuZZan: Efficient Sanitizer Metadata Design for Fuzzing appeared at ATC '20. Apart from my thesis, our initial type confusion detector, TypeSan: Practical Type Confusion Detection [3] appeared at CCS '16, and our process-aware least privilege approach for privilege manage APIs, PoLPer: Process-Aware Restriction of Over-Privileged Setuid Calls in Legacy Applications [36] appeared at CODASPY '19.

In summary, the core contributions of these papers are:

- Novel disjoint metadata structure based type confusion detector that reduces performance overhead up to 33.4 times, improves detection coverage up to 6 times, and discovers new type confusion vulnerabilities in real world applications.
- Novel inline metadata structure based type confusion detector that shows 0.81% overhead on the SPEC CPU2006, which is 11 times faster than HexType, and has 10 times higher detection coverage than Clang-CFI.

- Optimized sanitizers for fuzzing that improve memory safety violation detection throughput by 52% and find real-world memory safety violations up to 61% faster.

2 HEXTYPE: EFFICIENT DETECTION OF TYPE CONFUSION ERRORS FOR C++

Type confusion is the main attack vector to compromise modern C++ software like browsers or virtual machines. Generally, type confusion vulnerabilities are, as the name implies, vulnerabilities that occur when one data type is mistaken for another due to unsafe typecasting, leading to a reinterpretation of the underlying type representation in semantically mismatching contexts.

For instance, a program may cast an instance of a parent class to a descendant class, even though this is neither safe nor allowed at the programming language level if the parent class lacks some of the fields or virtual functions of the descendant class. When the program subsequently uses the fields or functions, it may use data, say, as a regular field in one context and as a virtual function table (vtable) pointer in another. Such type confusion vulnerabilities are not only wide-spread (e.g., many are found in a wide range of software products, such as Google Chrome (CVE-2017-5023), Adobe Flash (CVE-2017-2095), Webkit (CVE-2017-2415), Microsoft Internet Explorer (CVE-2015-6184) and PHP (CVE-2016-3185)), but also security critical (e.g., many are demonstrated to be easily exploitable due to deterministic runtime behaviors).

Previous research efforts tried to address the problem through runtime checks for static casts. Existing mechanisms can be categorized into two types: (i) mechanisms that identify objects through existing fields embedded in the objects (such as vtable pointers) [37–40]; and (ii) mechanisms that leverage disjoint metadata [41,42]. First, solutions that rely on the existing object format have the advantage of avoiding expensive runtime object tracking to maintain disjoint metadata. Unfortunately, these solutions only support polymorphic objects which have a specific form at runtime that allows object identification through their vtable pointer. As most software mixes both

polymorphic and non-polymorphic objects, these solutions are limited in practice — either developers must manually blacklist unsupported classes or programs end up having unexpected crashes at runtime. Therefore, recent state-of-the-art detectors leverage disjoint metadata for type information. Upon object allocation, the runtime system records the true type of the object in a disjoint metadata table. This approach indeed does not suffer from non-polymorphic class issues, because type information can be accessed without referring vtable pointers.

However, disjoint metadata schemes have to overcome two challenges: (i) due to C++'s low level nature it is hard to identify all object allocations and (ii) the lookup through this disjoint metadata table results in prohibitive overhead. Existing approaches with disjoint metadata precisely exhibit these drawbacks. Because it is difficult to handle all C++ language quirks imposed by developers, they only protect a small fraction of typecasts in practice. Due to the complexity of metadata tracking, existing approaches introduce prohibitive overheads (TypeSan [42] has up to 71.2% overhead for Firefox with a geometric mean of 30.8%; note that TypeSan already improves performance over CaVer [41]). Control-Flow Integrity (CFI) techniques [43–46] verify all indirect control-flow transfers within a program to detect control-flow hijacking. However, these techniques address the type confusion problem only partially if control flow is hijacked, i.e., they detect usage of the corrupted vtable pointer, ignoring any preceding data corruption. Similarly, vtable protection schemes [39, 40] protect virtual calls from vtable hijacking attacks but do not block type confusion attacks. Memory safety mechanisms [47–49] protect against spatial and temporal memory safety violations but incur prohibitively high overhead in practice. Also, these mechanisms do not protect against type confusion, e.g., they do not stop an `int` array of the correct size from being used in place of an object. Control-flow hijacking protection and memory safety are therefore orthogonal to type confusion detection. Type confusion may be used to cause a memory safety violation. Detecting type confusion allows earlier detection of security violations for these cases.

We propose HexType, a mechanism that protects C++ software from type confusion by making all casts explicit. Each cast in the source language (explicit or implicit, static or dynamic) is turned into a dynamic runtime check. HexType records the type of each object and specific casts are replaced with our instrumentation. We fundamentally address the challenges of earlier work by (i) increasing coverage of typecasting checks and (ii) drastically reducing overhead.

Our prototype implementation of HexType vastly outperforms state-of-the-art type confusion detectors, increasing coverage and often lowering overhead. Our reduced overhead is the result of novel optimization techniques and using an efficient type metadata structure. We leverage an analysis that identifies types that are used in typecasting, allowing us to remove tracing overhead for any objects that are never cast. For the type metadata structure, we design a two-layered data structure that combines a hash table (fast-path) and red-black tree (slow-path) in order to reduce object tracing overhead. Despite performance, our mapping scheme also overcomes limitations of existing work such as relying on fixed addresses for metadata which may run into compatibility issues if applications try to reuse the same addresses

To address the low coverage of related work, we developed allocation detectors that track reuse of pre-allocated memory space cases for new objects (through placement new) and transferring objects through `reinterpret_cast`. Additionally, HexType increases coverage for `dynamic_cast` and `reinterpret_cast` and goes beyond `static_cast` unlike all the previous works. In the case of `dynamic_cast`, HexType replaces the existing inefficient typecasting verification routine with a fast lookup using our metadata. HexType supports `reinterpret_cast` to increase object tracing coverage and find additional bugs.

Due to our increased coverage, we discovered four new type confusion vulnerabilities (which evaded previous approaches) in two widely-used open source libraries (Qt Base library and Apache Xerces-C++) during our evaluation. For the Firefox benchmarks, HexType increases coverage by 1.1 – 6.1 times compared to TypeSan with some increased performance overhead due to the vast increase in coverage. For

SPEC CPU2006 benchmarks with overhead, we show a 2 – 33.4 times reduction in overhead.

Our major contributions can be summarized as:

1. An open source type confusion detector with low overhead and high coverage (outperforming state-of-the-art detectors);
2. A novel optimization that greatly reduces the number of objects that need to be tracked (as much as 54% – 100% on SPEC CPU2006), thus reducing overhead;
3. Design of efficient data structures that use a fast-path ($\mathcal{O}(1)$ time complexity) for type information insertion and lookup (with a hit rate of 94.09% and 99.99% on the SPEC CPU2006 and 98.76% and 95.20% for Firefox respectively);
4. Robust allocation identification implementation that greatly increases coverage (1.1 - 6.1 times over TypeSan on Firefox) combined with also covering alternate casting methods such as placement new;
5. Discovery of four new vulnerabilities in QT Base library and Apache Xerces-C++;

2.1 HexType

2.1.1 Background

In this section, we provide background information on C++'s type system, various cast operations, and previous type confusion detection tools necessary to understand the design and implementation of HexType.

C++ Classes and Inheritance

C++ is an object-oriented programming language, with classes as the primary abstraction. Classes allow the programmer to define new types. A class can inherit from

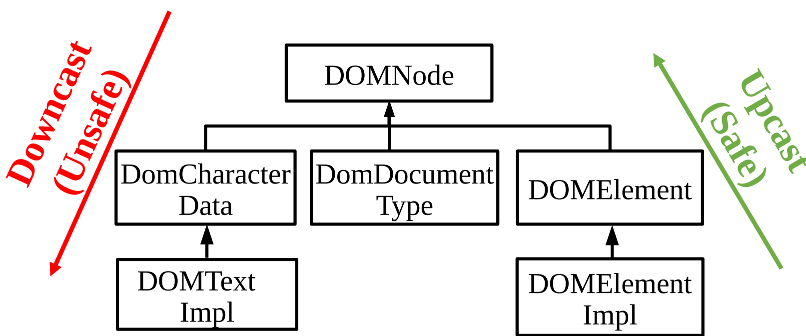


Figure 2.1.: Visualization of an example C++ type hierarchy, showing the directions of (safe) upcasting and (unsafe) downcasting.

```
...
class Ancestor { int x; };
class Descendant : Ancestor {
double y;
};

Ancestor *A = new Ancestor();
Descendant *D;
D = static_cast<Ancestor*>(A);
D->y; // error
...
```

Access scope
of *A

Ancestor::x

Access scope
of *D

Descendant::y

Figure 2.2.: A code example and diagram of a type confusion problem where an ancestor class is incorrectly accessed using a pointer to a descendent class. The static cast results in type confusion and accessing the field D→y results in a memory safety violation.

multiple ancestor classes. The descendent class has all the same members (methods and variables) as its ancestor(s) and optionally additional members defined in the descendent class definition.

In C++, a pointer of type A can be cast into a pointer of another type, type B. This effectively tells the compiler to treat the pointed-to object as being type B.

The crucial question is: when is a typecast safe? The answer depends on the type of the pointed-to object and the destination type (type B in the previous example). Focusing on casting between class types, the security objective of this work, casting from descendant class to ancestor class is always safe since the members of the descendant class are a superset of the members of the ancestor class. This operation is called upcasting. For example, as shown in Figure 2.1, if we visualize the type hierarchy with the ancestor class at the top and descendants at the bottom, moving up the

hierarchy (upcasting) is safe. On the other hand, downcasting, casting from ancestor to descendant, may not be safe if the ancestor misses any member of the descendant class. This is depicted in Figure 2.2. Such downcasting has been abused by attackers in a wide-range of popular C++ programs, which lead to complete compromises of an underlying system, as recently shown for, e.g., Google Chrome (CVE-2017-5023), Adobe Flash (CVE-2017-2095), Webkit (CVE-2017-2415), Microsoft Internet Explorer (CVE-2015-6184) or PHP (CVE-2016-3185).

C++ Cast Operations

The C++ syntax allows four different types of casts to meet different requirements of the developer. Each casting type performs unique casting operations, imposing non-trivial security implications. In the following, we provide detailed information on each casting type, particularly focusing on its security aspects in terms of type confusion issues.

The example in Figure 2.1 shows a cast using `static_cast`, but there are other cast in C++ and their details are important to this work. The other cast types we are concerned with are `dynamic_cast`, `reinterpret_cast`, and C-style typecasting.

```
static_cast<type>(expression)
dynamic_cast<type>(expression)
reinterpret_cast<type>(expression)
const_cast<type>(expression)
```

Static Cast A `static_cast` casts an object of type A to an object of type B. The check is executed purely at compile time and no runtime check is performed. Due to the static nature of this check, the runtime type of the object is not considered and the check is limited to check if the two types are compatible, i.e., there is a path in the type hierarchy from *expression*'s type and *type* that involves upcasting and/or downcasting.

While not incurring any performance overhead, the safety guarantees of static casts are limited. Therefore, the programmer is responsible that an object of the correct type is used, e.g., guaranteeing that the downcasted object is actually an object of the derived type. In practice, since it is challenging to figure out such compatibility at compile time, this has led to the unfortunate fact that type confusions are dominating vulnerabilities in modern C++ programs [50].

Dynamic Cast A `dynamic_cast` can safely convert types between classes in the same class hierarchy. Whereas `static_cast` only performs a compile time check, it performs an additional runtime check using heavy-weight metadata, Run Time Type Information (RTTI). As, in general, the dynamic runtime type of an object cannot be determined statically, `dynamic_cast` must leverage runtime type information such as RTTI. RTTI encodes all type related information, and a compiler generates this RTTI per type such that each type has its dedicated RTTI entry in a compiled binary. The RTTI entry essentially forms a recursive structure in that each RTTI entry points to another RTTI entry to represent the class hierarchy. A compiler further appends a reference to the RTTI entry at the end of each virtual function table, so that the RTTI entry can be retrieved at runtime using any virtual address pointing to an object. In other words, since the first field in an object is typically filled with a virtual function table pointer, `dynamic_cast` can find the RTTI entry given an object address using the virtual function table pointer. After locating the corresponding RTTI entry, `dynamic_cast` starts to recursively traverse RTTI to verify the casting correctness (i.e., that the types are compatible). If there is a path on the type hierarchy between *expression*'s type and the target type, then the types are compatible. The types are compatible whenever the type of expression is an descendant of *type* (upcast). The types can also be compatible when *type* is the exact type of the object pointed to by *expression*. If the casting is incorrect (i.e., the type of *expression* and *type* are incompatible), the cast fails in one of two ways:

- If *type* is a pointer type, it returns $NULL$.

- If *type* is a reference type, it throws a pre-defined exception (i.e., `std::bad_cast`).

Due to the design of `dynamic_cast`, its usage is strictly limited to polymorphic objects. As mentioned before, `dynamic_cast` relies on a virtual function table to locate RTTI, but the virtual function table is only present in polymorphic objects. Note that, given these limitations, `dynamic_cast` can only be used for polymorphic types. Thus, compilers simply generate a compile-time error if a `dynamic_cast` is used for a non-polymorphic type. Note that runtime errors are still possible.

Reinterpret Cast A `reinterpret_cast` converts between any two (potentially incompatible) types. It instructs the compiler to reinterpret the underlying bit pattern of the cast objects. Because it does neither create a copy nor perform any runtime check, a `reinterpret_cast` always incurs zero overhead. From the security standpoint, programmers are responsible to ensure the correctness of `reinterpret_cast` similar to the case in `static_cast`. Since `reinterpret_cast` only changes the object's type, it simply returns the same address. This behavior can cause problems for polymorphic classes or classes with multiple inheritance. For polymorphic classes, `reinterpret_cast` returns a pointer to an object with potentially the wrong vtable pointer as `reinterpret_cast` does not change the memory of the object. If the object uses multiple inheritance, then a pointer to a base class may have the wrong value (not a pointer to the object itself) [51]. However, if the exact source object type information is known then `reinterpret_cast` can be used to: (1) efficiently construct an object without executing the constructor (reusing an old object of the same type) and (2) restoring the actual type if a function returns a `void*` pointing to an object.

Const Cast A `const_cast` drops cv-qualifier (i.e., const or volatility) from an object specified in the expression. Unlike previously mentioned static and dynamic casts, `const_cast` does not impact type confusion issues because type hierarchies are not involved in this case. `const_cast` still may introduce security issues (5.2.9/11 in ISO/IEC 14882 [52]) if it is used in the wrong context—a read-only object has been accidentally const cast, and thus overwrites such a write-protected object. These

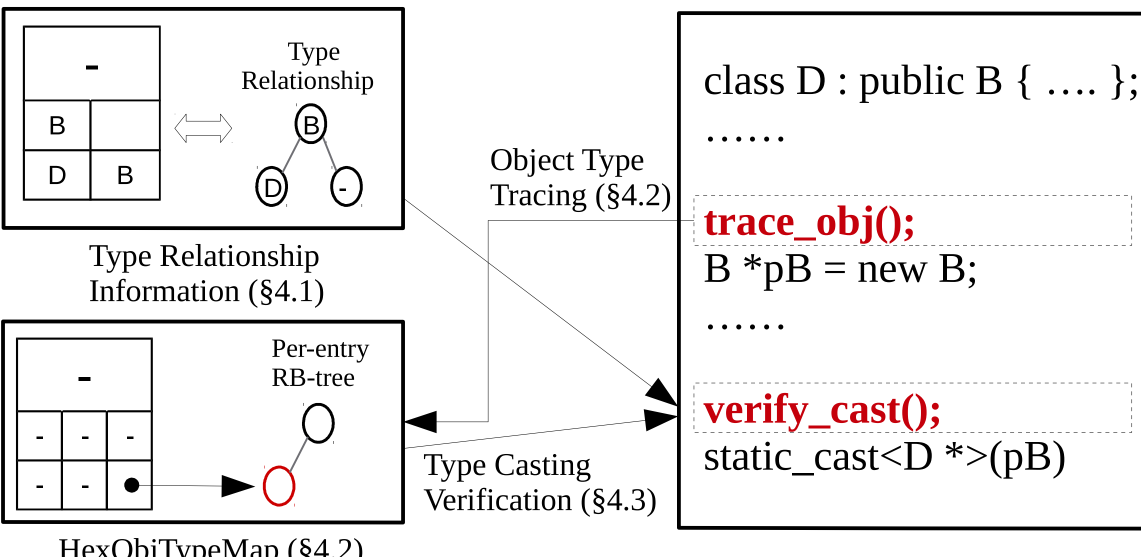


Figure 2.3.: A system overview of HexType. HexType consists of several modules that analyze type relationship information and insert object tracing and typecasting instrumentation to verify typecasting operation.

issues can be addressed using other memory safety techniques [30, 30, 32, 32] through enforcing per-object write protection. Protecting const casts is therefore orthogonal to the scope of this paper.

C-style Typecast Although C-style casts are discouraged in C++ programs, compilers allow them to keep backward compatibility. More precisely, if C-style casting (5.4 in ISO/IEC 14882 [52]) is encountered, the compiler translates the cast into a sequence of casts: (i) `const_cast`, (ii) `static_cast`, and (iii) `reinterpret_cast`. In other words, compilers try to cast the objects using the sequence of casts above and use the result of the first cast that succeeds without a compilation error. This in fact implies that, from the standpoint of detecting type confusion issues, it is no different from handling the above three cast types as C-style casting will finally be translated into one of them.

Defenses against type confusion

Type confusion is a pressing problem and several mechanisms have been proposed to detect and protect against type confusion. As mentioned earlier, the existing defenses can be grouped into two categories: (i) those based on identifying objects based on existing fields embedded in the object themselves (such as vtable pointers) [37–40]; and (ii) those based on disjoint metadata [41, 42].

CaVer [41] uses disjoint metadata for *all* allocated objects to support non-polymorphic classes without blacklisting. CaVer is the first typecasting detection tool (based on disjoint metadata) that can verify type-casting for non-polymorphic objects. However, CaVer suffers from both security and performance issues — low safety coverage on castings and high runtime overhead.

TypeSan [42] reduces the performance overhead by a factor of 3 – 6 compared to CaVer and increases detection coverage by including C-style allocation (e.g., `malloc`). However, the overhead of both disjoint metadata approaches is still high due to inefficient metadata tracking, e.g., tracking most live objects. Also, while increasing coverage compared to CaVer, TypeSan still has an overall low coverage rate. Especially, TypeSan has 12 $\sim$ 45% coverage rate for Firefox. These limitations motivated us to design HexType, which overcomes the aforementioned limitations — namely reducing per-cast check overhead, increasing coverage, and providing additional features.

2.1.2 Threat Model

Our threat model assumes that the underlying application is benign but contains a type confusion error that an attacker can find and exploit. The primary goal of our defense mechanism is to prevent such type confusion attacks. Our defense mechanism automatically detects such exploitation attempts, avoiding any negative security ramifications. We further assume that the attacker may read arbitrary memory, and thus our detection mechanism is designed not to rely on information hiding or randomiza-

tion. Attacks not based on type confusion, including control-flow hijacking, integer overflow, and memory corruption, are out of scope and these can be protected by other security hardening techniques. We assume that our instrumentation cannot be removed by the attacker, i.e., our instrumented code is on a non-writable page. The underlying operating system, program loader, and system libraries are in the Trusted Computing Base (TCB).

2.1.3 HexType Design and Implementation

HexType is a Clang/LLVM-based type confusion detector for C++ programs. During compilation of a target program, HexType generates a HexType-hardened program. During runtime, if HexType detects a type confusion error, the program is terminated with a detailed bug report.

Figure 2.3 illustrates an overview of HexType. Given the source code as input, HexType generates a type table containing all type relationship information (section 2.1.3) and, at runtime, information about the true types of each allocated object is collected in the object mapping table (section 2.1.3). HexType verifies the correctness of each cast using both the type relationship information and object mapping table (section 2.1.3). HexType leverages a set of optimization techniques to reduce performance overhead during the above processes (section 2.1.3).

Type Relationship Information

In order to verify typecasting operations, HexType needs to know a valid set of destination types that can be cast from a given source type. Note that compilers keep this information readily available during compilation to check the validity of casts statically, but such checks are inherently limited as the true source type of an object is only known at runtime. C++ applications generally do not keep explicit information about the type hierarchy. This subsection describes how HexType generates and

maintains a hierarchical type information for executables and shared libraries. We call this information type table.

During compilation, HexType extracts all type relationship information and prepares metadata for each type. For example, as shown in Figure 2.1, for the type `DOMElementImpl`, HexType first collects all types that are allowed to be cast (i.e., `DOMElement` and `DOMNode`), each of which is basically a parent class of `DOMElementImpl`. Instead of simply storing a type name in the type table, HexType stores a string hash of the type name to avoid expensive string match operations, enabling O(1) comparisons. HexType exports, per type, a list of hash values as a global variable during the compilation, allowing other libraries to reuse this information. These lists of hash values are sorted to efficiently search value from the target list using binary search during runtime type casting verification. HexType generates one such global variable per type.

```
DOMElementImpl: H(DOMElement), H(DOMNode), ...
```

In order to provide compatibility, HexType allows and the type table includes phantom classes. A phantom class is a parent-child relationship where the data layout of the child is equivalent to the data layout of the parent. HexType allows downcasts from such a child to the parent as such phantom classes are frequently used in practical environments to support interoperability between C and C++.

To manage the type table efficiently, HexType only records each type's relationship information once, following the one definition rule (ODR) [52] of the C++ standard. According to this rule, the type definition of each object must be identical (each object's parent information is always the same) among all source code which will be merged. Therefore, each type will have a uniquely identical list of hash values among all source codes except for phantom classes. Since each object can have a derived class as a phantom class and the set of the phantom classes cannot be determined when each object type is defined (we also have to rely on information from each object's derived class defined site), HexType only needs to update this phantom class information.

Object Type Tracing

In order to verify typecasting operations at runtime, HexType needs to locate the type information based on the underlying object identified by the source pointer address in the casting operation. Unlike `dynamic_cast`, HexType does not utilize RTTI to retrieve type information due to the following limitations of RTTI: (i) RTTI only provides type information for polymorphic objects (not supporting typecasting verification of non-polymorphic objects); (ii) RTTI incurs expensive typecasting verification costs due to its recursive structure; and (iii) RTTI significantly blows up the size of the compiled binary.

For these reasons, HexType designs a new set of techniques, which aims at maximizing security coverage and minimizing performance overhead. In the following, we first describe how HexType captures the underlying memory semantics with respect to the type information. HexType systematically identifies all object allocation sites, which significantly elevates the coverage for typecasting operations (section 2.1.3). Next, we illustrate how HexType maintains such memory semantics at runtime. In order to perform efficient lookup operations, HexType employs a new data structure, type table, which supports both a fast-path for performance efficiency and a slow-path for completeness (section 2.1.3).

Tracing Object Type Allocation

The C++ type system is not strongly constrained and thus developers can easily change the object type at runtime as required. This flexibility, though it is one of the main reason of C++'s popularity, introduces several challenges when tracking type information. More precisely, HexType must identify the correct type information imposed to certain runtime memory objects, but dynamic type changes complicate the identification processes.

HexType comprehensively identifies all the sites that assign types, which can be generally categorized into the following two cases depending on when the type as-

signment is performed—(1) at the time of creating an object and (2) at the time of transferring an object. The first case includes the well known new operator which allocates object memory space through typical system memory allocator (i.e., `malloc()`) and initializes the object by invoking its associated constructor function. The first case also includes *placement new*, which reuses specified memory space and simply invokes the constructor for initialization. For these type allocation sites at object creation time, HexType registers the type of the object in the type table by passing the type information and the base pointer to the registration function. The runtime library function updates the type table with this information

We describe more details how HexType maintains information in section 2.1.3.

The second case of type assignments happen while hard-copying objects that have already been constructed. In C++, it is common to copy or move memory objects in memory space for, e.g., object marshaling or when passing objects between allocation spaces. Once the memory object is relocated in memory space, a developer is responsible to reassign the type of underlying memory objects. Developers can rely on move or copy operators in C++ if the underlying object is a C++ class object constructed through new or placement new operators. Alternatively, they can explicitly specify the type of underlying memory objects using `reinterpret_cast`. This second case is commonly used to work around system constraints. For example, when an object is marshaled and unmarshaled to pass it between different components, `reinterpret_cast` can efficiently construct an object without explicitly executing the class constructor again. To handle `reinterpret_cast`, HexType instruments `reinterpret_cast` to call a runtime function with two pieces of information: (i) destination type and (ii) source address information. In the runtime library function, HexType inserts this information into the type table only if there is no matching entry with `reinterpret_cast`'s source address.

For example, Figure 2.4 shows how aligned_storage creates and accesses objects. In the initial step, aligned_storage creates uninitialized memory blocks (line 5). In this

```
template<class T, std::size_t N>
class static_vector
{
 // properly aligned uninitialized storage for N T's
 size_t Size = sizeof(T);
 size_t Align = alignof(T);
 typename std::aligned_storage<Size, Align>::type d[N];
 ......

 public:
   template<typename ...Args> void insert(Args&&... args)
   {
     ......
     // Create an object using placement new
     new(d+m_size) T(std::forward<Args>(args)...);
     ......
   }

   const T& operator[](std::size_t pos) const
   {
     // Access an object using repinterpret_cast
     return *reinterpret_cast<const T*>(d+pos);
   }
   ......
 };
```

Figure 2.4.: Code example for std::aligned_storage using placement new and `reinterpret_cast` to manage type allocation.

uninitialized storage, the objects are created using placement new (line 13). Then, we can access the allocated objects using `reinterpret_cast` (line 20).

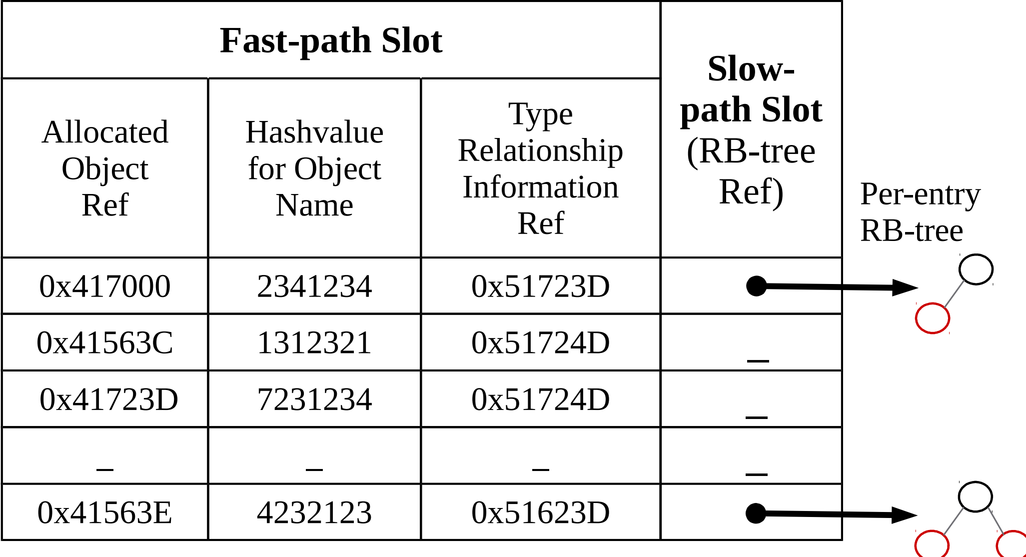
In fact, previous work including UBSan, CaVer, and TypeSan all fail to generally handle type assignment sites. In the case of UBSan, it cannot capture the type information of non-polymorphic objects as it has to rely on RTTI, resulting in unexpected crashes at runtime. In the case of CaVer and TypeSan, they only consider new operator as type assignment sites and thus they miss all other assignment sites mentioned above. As we will clearly demonstrate in the evaluation section, HexType showed 1.1 – 6.1 times higher coverage on Firefox benchmarks compared to TypeSan.

Mapping Objects to type table

HexType maintains an object mapping table, which maps runtime objects to its associated type information in the type table. More specifically, a key in the object mapping table is an object address and its mapped value is an address pointing to the associated entry within the type table. It is performance critical for HexType to efficiently design this object mapping table, because this mapping process through object mapping table is performed every time HexType attempts to verify the typecasting operations.

We found various object tracking methods in previous works [41,42]. TypeSan [42] uses a memory shadowing scheme to track global, heap, and stack objects. However, the TypeSan memory shadowing scheme has three limitations: (i) TypeSan uses a fixed address for the metadata table (to enable faster lookups) which may result in compatibility problems if applications reuse the same address, e.g., due to ASLR, which we observed in practice;

(ii) TypeSan only updates objects in the "object to type" mapping table when objects are allocated (it does not delete information when an object is deleted from memory). Therefore stale metadata can create additional problems; (iii) TypeSan's memory shadowing scheme uses more memory resources compared to other non-



Fast-path Slot			Slow-path Slot (RB-tree Ref)
Allocated Object Ref	Hashvalue for Object Name	Type Relationship Information Ref	
0x417000	2341234	0x51723D	•
0x41563C	1312321	0x51724D	–
0x41723D	7231234	0x51724D	–
–	–	–	–
0x41563E	4232123	0x51623D	•

Figure 2.5.: A snapshot example of object mapping table, showing how it maps an object using a combination of fast-path and slow-path slots. When HexType looks up type information, an object address is used to obtain the reference to the corresponding object mapping table entry. HexType first matches the fast-path slot. If not present in the fast-path slot, HexType then searches the corresponding red-black tree to find type information (slow-path) and updates the fast-path accordingly.

memory shadowing schemes. CaVer [41] uses a red-black tree to keep track of global and stack objects. However, overhead becomes prohibitive for, e.g., stack objects, as stack objects incur frequent insertions and deletions. Since a red-black tree generally shows $\mathcal{O}(logN)$ time complexity to delete, insert, and search.

Toward this end, HexType leverages a new data structure to reduce performance overheads in mapping operations. The key insight for object mapping table is that some objects are accessed much more frequently than others. We therefore designed a data structure that splits object lookup into a fast pass using a hash table and a slow path using a red-black tree, see Figure 2.5. The first level of our data structure is a hash table. We use the object's address and a simple hash function to locate the entry for a given object. Each hash table entry holds two slots: (1) the fast-path slot for the least recently cast object, which holds the reference to the object (to check if the object matches), the hash of the object's type, and the reference to

object's type relationship information (collects all destination types that are allowed to be cast) and (2) the slow-path slot, which holds a reference to a per-slot red-black tree maintaining a complete set of objects that map to the hash table entry. In other words, once HexType locates a hash table entry, it simply reuses the value in the fast-path slot if the object's address in the fast-path slot matches. Otherwise, HexType walks through the red-black tree pointed by the slow-path slot to address collisions. Whenever a lookup in the red-black tree is performed, the fast-path is updated with the most recent object. As a result, our mapping scheme with object mapping table imposes $\mathcal{O}(1)$ time complexity for fast-path accesses and $\mathcal{O}(logN)$ for slow-path accesses (where N is the number of values in the per-slot red-black tree). In the SPEC CPU2006 C++ benchmarks, our approach uses the fast-path 99.68% of time to update metadata and 100% of the time to lookup information from the type table. We demonstrate that these design choices for the object mapping table are reasonable in the evaluation section in subsection 2.1.5.

Type Casting Verification

We now describe the final step of HexType, typecasting verification, which checks the safety of casting. HexType instruments typecasting operations with additional verification code at compile time. At runtime, this instrumentation locates the object's true type information in the type table and then compares the target type with the expected type at the cast site to determine if the cast is legal.

HexType instruments all type casting operations related to type confusion issues. As described in section 2.1.1, these include `static_cast` sites, where its casting operation performs downcasting. More precisely, HexType instruments additional code invoking a runtime verification function while passing necessary information with respect to casting verification (i.e., a base object pointer subjected to casting and a hash value of a destination type).

```
class Base1 { ... };
class Base2 { ... };

// multiple inheritence
class Derived: public Base1, public Base2 { ... };

Derived obj;
Derived* dp = &obj;

// indicates the Dervied's Base2 object
Base2* b2p = dp;
// static_cast restores the original pointer value
Derived* dps = static_cast<Derived*>(b2p);
// reinterpret_cast perserves the new pointer value
Derived* dpr = reinterpret_cast<Derived*>(b2p);
```

Figure 2.6.: An example of how `reinterpret_cast` results in a type confusion problem.

Additionally, HexType also verifies `reinterpret_cast`. As mentioned in section 2.1.1, `reinterpret_cast` forces the casting operation by copying the memory bits of a pointer value even though casting types are not compatible. Thus, this operation is security critical if misused. For example, as shown in Figure 2.6, since `reinterpret_cast` simply returns the same unchanged address (line 15), a pointer to a base class points to a semantically different object, which results in access to an unexpected memory area. In other words, `reinterpret_cast` does not properly adjust the pointer according to the class hierarchy (line 5) as it simply hard-copies a to-be-cast value, compared to `static_cast` which adjusts the pointer.

Once a runtime verification function is invoked at runtime, HexType first locates the object mapping table. Given the base pointer address of an object, HexType computes the hash index within the object mapping table, which returns a reference to the corresponding type table walking through either fast-path or slow-path (section 2.1.3). Using this type table as well as the provided destination type information, HexType reasons about whether the underlying object can be indeed a sub-object of the destination type such that the casting itself is correct in the end. If HexType detects type confusion at runtime, it displays a detailed report that includes allocated object type, expected object type, and the type casting location. This information allows the developer to triage the type casting issue quickly.

Optimization

Type casting verification supported by HexType may impose non-negligible performance overhead as it involves additional computation. In order to make HexType a truly practical security tool, we implement a set of performance optimization techniques, namely only tracing unsafe objects, only verifying unsafe casting, and efficient dynamic casting.

Only Tracing Unsafe Object HexType only traces type information on potentially unsafe objects and does not trace safe objects. We define T as a safe object type if

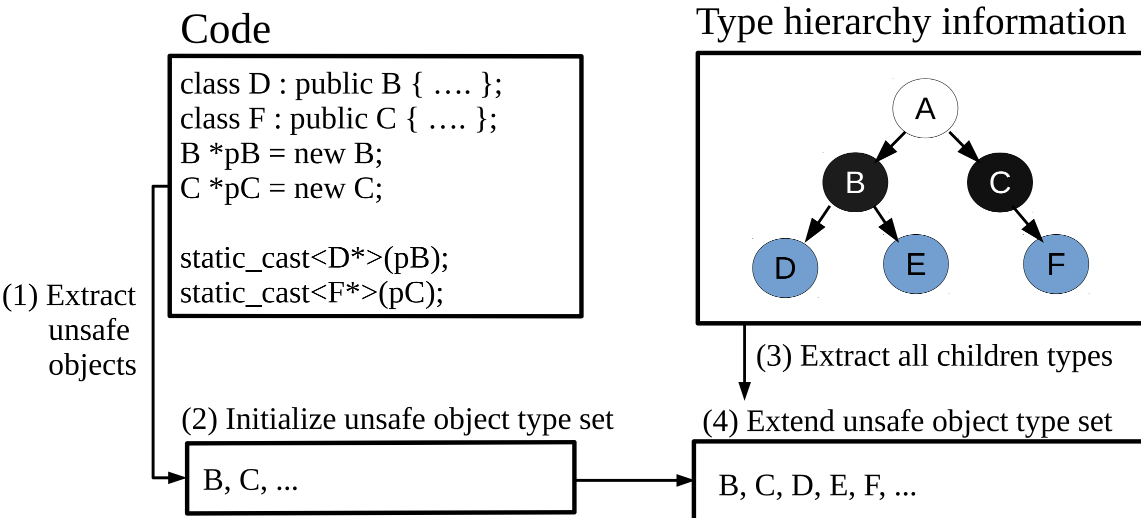


Figure 2.7.: An example of how HexType creates a potentially unsafe object type set. In the example, we assume that objects of type B and C are typecast. HexType will identify these potentially unsafe types and all its children types as unsafe.

and only if T is never subject to typecasting. T is a potentially unsafe object type otherwise. Since safe object types will never be used for casting at all, HexType does not need to keep track of them to check casting validity. We assume that the source pointer of a safe object always references an object of the correct type as no casting operation in the program exists that breaks this assumption. As illustrated in Figure 2.7, HexType performs the following two steps to identify unsafe objects. First, it identifies a typecasting-related object set, which can be used for typecasting operations. HexType identifies all type information both at the casting site and for the cast object. An object that is cast can be of type X or any of the child classes of type X. The type casting site therefore must accept all possible subtypes. Next, when instrumenting object allocation sites, HexType selectively instruments allocation only for typecasting-related objects.

While evaluating HexType, we found that tracking stack objects is the most critical performance bottleneck. Thus, considering allocation characteristics of stack objects, we apply a special optimization scheme to conservatively distinguish safe stack objects from unsafe stack objects.

First, we apply CaVer's optimization technique which is based on the observation that the lifetime of a stack object can be relatively well defined with respect to a set of functions the object is active — a function (that the subjected stack object is declared) and all of its callee functions, if there are no out-going indirect calls. Thus, only if there are no out-going indirect calls, we perform an escape analysis for the set of those functions so as to ensure that any reference to the stack object never leaves the analyzed functions. Further more, if there are no typecasting operations within these clustered and side-effect free functions, then the analyzed stack objects will never be used for typecasting. In this case, it is truly a safe stack object that does not need to be tracked at runtime.

We apply a more fine-grained analysis for functions that did not pass the previous check: (i) we check whether each stack object in the function is a local variable using SafeStack which is a component of CPI/CPS [46], since SafeStack supports local variables detection and (ii) if these local stack objects are not used for any typecasting operation within this function, we do not need to trace these stack objects.

Only Verifying Unsafe Casting Clearly HexType does not need to perform runtime verification for a casting operation if it can be proven safe during compilation. We call such a provably safe cast operation a *safe casting*, and *unsafe casting* otherwise. Since HexType supports runtime casting verification, we can leverage an optimization that relies on an imprecise yet conservative static analysis to distinguish these two categories. In other words, given a casting operation, HexType determines if it is safe casting only if HexType can be completely certain at compile time. If HexType cannot determine it is safe in a compile time, HexType simply considers it unsafe casting and falls back to a runtime check.

HexType leverages a conservative backward dataflow analysis to identify safe casting. Starting from a casting site, HexType reasons about type information of an underlying object, i.e., how the underlying object has been allocated. To answer this question, we perform an inter-procedural use-def chain analysis, where the use point is defined as a casting site and the def point is defined as any object allocation sites.

```
class T : public S { ... };

void safe_casting_ex() {
  S test1;
  S test2[1000];

  // safe casting : always cast from class S
  // (case 1)
  static_cast<T*>(&test1);

  // (case 2)
  static_cast<T*>(test2);

  // (case 3)
  S *local_obj_ptr = &test1;
  static_cast<T*>(local_obj_ptr);
}

void unsafe_casting_ex() {
  // unsafe casting : type is hard to dertermine
  S* obj_ptr = external_func();
  static_cast<T*>(obj_ptr);
}
```

Figure 2.8.: An example for safe and unsafe casting. The three examples (line 9, 12, and 16) are all safe casting. For the first two examples, each typecasting operation obtains the object address using the address-of operator and the array name. In the third example (line 14), we can simply determine the object type using use-def chain analysis. The last example (line 22) is an unsafe casting case when we cannot track the object type (i.e., external function).

For example, as shown in Figure 2.8, if the source of typecasting operation uses the address-of operator(&) or array name directly to get the address of the object, HexType can easily determine the source object type and verify the typecasting operation at compile time. Also, we can predict the object type through the use-def chain analysis. In these cases, we can remove HexType's typecasting verification instrumentation and verify typecasting operations during compile time. However, if we cannot determine the source type, HexType will again fall back to a runtime check.

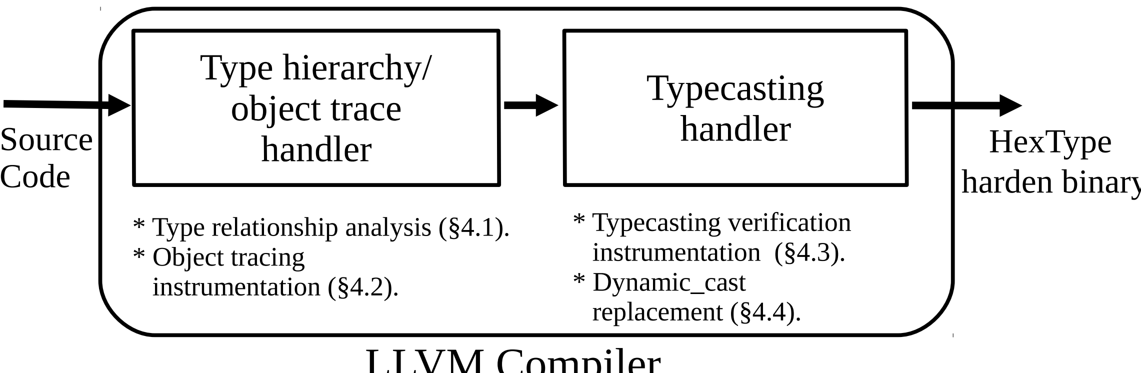


Figure 2.9.: Overview of HexType's implementation. HexType consists of several compiler passes in clang and LLVM that insert object tracing and typecasting instrumentation and a corresponding runtime library.

Efficient Dynamic Casting Since HexType offers efficient runtime casting verification, the existing `dynamic_cast` can be optimized accordingly. HexType therefore replaces each `dynamic_cast` with our fast lookup. In order to preserve the runtime semantics of `dynamic_cast` as dictated by the C++ standard, HexType takes additional steps in response to an incorrect casting detected in runtime. As described in section 2.1.1, HexType returns NULL for a pointer-typed casting and throws an exception for a reference-typed casting. This optimization can be especially useful if applications heavily rely on dynamic casting.

2.1.4 Implementation

We have implemented HexType, as shown in Figure 2.9, based on the LLVM Compiler infrastructure project [53] (version 3.9.0). The HexType implementation consists of 4,677 lines of code that we added to Clang, an LLVM Pass, and our compiler-rt runtime library. HexType's LLVM Pass (i) creates type relationship information, and (ii) instruments allocations of unsafe objects to record allocated object type information into our object mapping table. Also, we modify Clang to (i) instrument all downcast sites (the pointer type of casting operation is one of the parent objects of destination type), and (ii) handle `dynamic_cast`, `reinterpret_cast`, and placement new.

At runtime, the instrumentation invokes HexType's runtime library functions to update object allocation information into object mapping table, and verifies typecasting operation using type relationship information and object mapping table.

2.1.5 Evaluation

In this section, we evaluate HexType focusing on following aspects: (i) the detection coverage (section 2.1.5); (ii) newly discovered vulnerabilities by HexType (section 2.1.5); (iii) the efficiency of object mapping table (section 2.1.5); and (iv) runtime overhead (section 2.1.5).

Experimental Setting All evaluations were performed on Ubuntu 16.04.2 LTS with a quad-core 3.60GHz CPU (Inter i7-4790), 250GB SSD-based storage, 1TB HDD, and 16GB RAM.

Evaluation Target Programs We have applied HexType to the following programs: all seven C++ benchmarks from SPEC CPU2006 [54] and Firefox [55]. For Firefox, we use Octane [56] and Dromaeo [57] benchmark suites. Moreover, in order to compare HexType with previous work, we applied TypeSan as well, and ran these programs under the same configuration. For CaVer, we use the numbers from the paper since CaVer was developed almost three years ago and we encountered compatibility issues with the current test environment and software (i.e., Firefox).

Coverage on Typecasting

One of the primary goals of HexType is in increasing the typecasting coverage such that HexType can ensure that all different typecasting operations are correctly performed. To evaluate typecasting coverage, we counted how many typecasting operations were verified at runtime (shown in Table 2.1). We used two different versions of HexType in this experiment, where each version either turned off or on the optimization techniques presented in section 2.1.3 (denoted as HexType-no-opt

Table 2.1.: The evaluation of typecasting verification coverage against SPEC CPU2006 and browser benchmarks. Columns with % present a coverage ratio and columns with × present a coverage improvement ratio (i.e., HexType's coverage divided by TypeSan's coverage).

	# of casting	**TypeSan**	**HexType -no-opt**		**HexType**	
		%	%	×	%	×
omnetpp	2,014m	100	100	1	100	1
xalancbmk	283m	89.5	99.8	1.1	99.8	1.1
dealII	3,596m	100	100	1	100	1
soplex	209k	100	100	1	100	1
ff-octane	623m	12	73.3	6.1	56.5	4.7
ff-drom-js	4,229m	23	80.1	3.5	59.4	2.6
ff-drom-dom	10,786m	45	88.8	2	54.9	1.2

and HexType, respectively). For TypeSan, we referred to the evaluation numbers presented in the paper [42].

For SPEC CPU2006, HexType verifies almost all typecasting operations — 100% for omnetpp, dealII, and soplex, and 99.8% for xalancbmk. Compared to TypeSan, HexType improves the coverage number on xalancbmk (i.e., improved from 89% to 99.8%). This is because xalancbmk heavily uses placement new to allocate objects, for which TypeSan looses information about the object at runtime. Thus TypeSan fails to resolve type information associated with such objects. However, as described in section 2.1.3, HexType correctly handles these new operator allocations, which significantly raised the coverage ratio.

For Firefox, depending on the benchmark suite, HexType successfully covers typecasting operations: ranging from 73% to 88% with HexType-no-opt; and ranging from 54% to 59% with HexType. During our evaluation, we found that HexType's coverage rate drops after applying our optimizations due to interactions with Firefox's complex object allocation patterns and how our optimizations handle and track allocations in LLVM/Clang. While we are investigating and plan to fix this issue in the future, HexType with optimization still shows better coverage rate than TypeSan. Most of the missing type casts in xalancbmk and Firefox result from application-specific allocation patterns. More specifically, Firefox creates a custom storage pool (typed as an array of char), and manipulates the pool using memcpy or direct object initialization (e.g., `data.key = key; data.index = index; ..`). Xalancbmk also uses a special storage pool (SerializeEngine) that manages objects directly without calling memory allocation functions. As these allocation patterns cannot be detected by HexType during the instrumentation phase, HexType cannot track runtime object types that are allocated through these patterns. Handling these missing allocations is challenging. A naive approach would trace the custom storage pool (allocated as char array) and its low-level allocation patterns using memcpy or direct object initialization. This would unfortunately result in high overhead. Alternatively, we propose to modify the few locations in Firefox and annotate the object allocation accordingly.

Although this coverage ratio in Firefox may not be as impressive as HexType's result of the SPEC CPU2006 benchmarks, we emphasize it is significantly improved from the state-of-the art tool, TypeSan. TypeSan only covered 27.75% of Firefox's typecasting on average, 52.98% and 29.18% less than HexType-no-opt and HexType, respectively, highlighting HexType's advantage in identifying allocation sites (section 2.1.3).

Newly Discovered Vulnerabilities

During the course of evaluating HexType by running the set of target programs, we discovered four new type confusion vulnerabilities. In particular, HexType reported four vulnerable cases in the Qt base library while evaluating Wireshark and Apache Xerces-C++, all of which have been confirmed and patched by the corresponding developer communities. For Apache Xerces-C++, HexType found two new vulnerabilities. These vulnerabilities occurred due to type confusion issues between `DOMNodeImpl` (indicated by `DOMNode` type pointer) and `DOMTextImpl`. Since the DOMNodeImpl object is allocated using placement new from a pre-allocated memory pool previous approaches cannot trace these objects. Therefore, these vulnerabilities were not detected by previous schemes such as CaVer or TypeSan.

In addition, HexType found two new vulnerabilities in the Qt-based library. The Qt team already patched our reported type confusion bugs [58]. HexType reported type confusion issues when Qt performs a casting from `QMapNodeBase` (base class) to `QMapNode` (derived class). Since QMapNode is not a subobject of QMapNodeBase, it violates C++ standard rules 5.2.9/11 [52] (down casting is undefined if the object that the pointer to be cast points to is not a subobject of down casting type) and causes undefined behavior.

These new vulnerabilities discovered by HexType clearly demonstrate the security advantage of HexType, especially compared to other previous work including TypeSan and CaVer. We would like to further point out that these new type confusion vulnerabilities were discovered only with basic benchmark workloads. In the future,

we plan to run HexType under a fuzzing framework such as AFL [59]. to discover more security critical vulnerabilities related to type confusions.

Efficiency of Object Tracing

Recall that the key runtime functions that HexType performs are (1) keeping track of object types (at the time of object allocation) and (2) looking up an object type (at the time of type casting). As described in section 2.1.3, we designed object mapping table to efficiently handle these operations leveraging both a fast-path and a slow-path. Therefore, the performance efficiency of object mapping table clearly relies on the hit ratio of the fast-path (i.e., the number of operations that only access the hash table) such that HexType does not need to consult the slow-path (i.e., accessing not only the hash table but also the corresponding red-black tree) in most cases.

Table 2.2 lists the fast-path hit ratios while running the set of evaluation target programs. Overall, most of operations showed high fast-path hit ratios, ranging from 98.820% and 99.999% to update object mapping table and from 94.099% and 100% to lookup object mapping table. This high fast-path hit ratio was also maintained when HexType was running large-scale programs such as Firefox, which creates more than 37,000 million objects at runtime. This result implies that the design decision of the object mapping table is efficient enough to support a wide range of programs, which in turn significantly helped HexType to reduce runtime impact.

Performance Overhead

To analyze performance impacts imposed by HexType, this subsection measures performance overhead in terms of runtime speed. Table 2.4 shows the performance overhead on the SPEC CPU2006 and Firefox, handling placement new and `reinterpret_cast`. For all seven C++ benchmarks in SPEC CPU2006, HexType outperformed previous work in all cases. This is largely because of HexType's optimization algorithms (section 2.1.3) as well as object mapping table designs (section 2.1.3).

Table 2.2.: The number of traced objects and its fast-path hit ratio when HexType lookup/update these objects into our the object mapping table.

	allocated objects			**fast-path hit ratio (%) (update)**	**fast-path hit ratio (%) (lookup)**
	stack	**heap**	**global**		
omnetpp	1m	478m	601	99.999	100
xalancbmk	3,150m	45m	3,098	99.998	99.999
dealII	497m	283m	200	99.988	100
soplex	21m	639m	197m	99.691	100
ff-octane	593m	7m	125k	98.820	98.649
ff-drom-js	2,875m	11m	125k	99.645	98.426
ff-drom-dom	34,900m	607m	125k	99.706	94.099

Table 2.3.: The number of safe objects identified by HexType's optimization algorithm. HexType does not keep track of these safe objects.

	# of object	safe casting related object (%)	safe stack objects (%)	total safe objects rate (%)
omnetpp	480m	54.76	0.107	54.767
xalancbmk	3,196m	99.42	3.50	99.42
dealII	781m	83.81	51.07	83.81
soplex	858m	97.75	0.76	97.87
povray	6,550m	100	0.18	100
astar	28m	100	1.31	100
namd	2m	100	0	100
ff-octane	600m	42.69	1.96	44.11
ff-drom-js	2,491m	39.99	1.42	40.26
ff-drom-dom	37,538m	21.33	0.95	21.54

Table 2.4.: SPEC CPU2006 and browser benchmark performance overhead for CaVer, TypeSan, and HexType. The $\times_1$ column denotes the ratio between CaVer and HexType and $\times_2$ denotes between TypeSan and HexType.

	CaVer	**Type San**	**HexType**		
	%	**%**	**%**	$\times_1$	$\times_2$
omnetpp	NA	49.13	9.69	NA	5.1
xalancbmk	29.6	41.35	1.25	23.7	33.1
dealII	NA	78.23	13.13	NA	6
soplex	20.0	1.16	0.76	26.3	1.5
astar	NA	0.36	0.34	NA	1.1
namd	NA	-0.37	-0.37	NA	1
povray	NA	26.73	0.8	NA	33.4
ff-octane	45	19.37	30.87	1.5	-1.6
ff-drom-js	40	25.18	25.89	1.5	-1.03
ff-drom-dom	55	97.15	126.03	-2.3	-1.3

To clearly understand these, Table 2.3 reports how many objects HexType identified as safe objects. With the help of the optimization algorithm, HexType was able to dramatically reduce the number of objects to be traced — reduced from 83% to 100% of tracing for all cases except omnetpp. For omnetpp, the number of casting

related classes (unsafe objects) is higher than other cases. However, we can reduce almost 54% object of the tracing overhead. Interestingly, out of the seven SPEC CPU2006 C++ benchmarks that we ran, povray, astar, and namd do not perform any typecasting operation that HexType has to verify at runtime. This implicates that HexType will have zero overhead for these cases since there are no object tracing and typecasting operation. In comparison, TypeSan imposes 26.73% overhead for povray.

In the case of omnetpp and dealII, HexType has shown significantly better performance due to HexType's optimization on replacing `dynamic_cast` (section 2.1.3). This optimization technique can show strong performance improvements, particularly for the applications heavily relying on `dynamic_cast`. We analyzed programs in our evaluation set, and found that two SPEC CPU2006 C++ benchmarks, dealII and omnetpp, perform a huge number of `dynamic_cast`, 206 M and 47 M number, respectively. Therefore, we replaced `dynamic_cast` in our verification routines which reduced the dealII's performance overhead by 4%.

For Firefox, HexType showed similar or higher overhead than TypeSan. Note that, when assessing performance, HexType vastly extends coverage compared to TypeSan (past the differences in coverage).

Moreover, while HexType reduced object tracing by nearly 52 – 100% in SPEC CPU2006, it only reduced the number of traced objects by about 21 – 44% in Firefox. We also suspect this is because of the Firefox's runtime characteristic — almost all objects in Firefox, as shown in Table 2.2, are allocated on the stack. We note that TypeSan's object mapping scheme comes with a security rist as it never removes object type information. As a result, if the stack location of a former properly allocated object is used in a casting operating, it may be interpreted as a valid object. However, since HexType properly deletes those information when the stack returns, HexType does not suffer from these security issues.

2.1.6 Discussion

Coverage This paper extends the coverage of type confusion detection, particularly in `dynamic_cast` and `reinterpret_cast`. In the future, it is also possible to handle more subtle type confusion issues such as `const_cast`'s undefined behavior or union's type confusion problem. In the case of `const_cast`, it is only safe if we are casting a variable that was originally non-const. Otherwise, a program may modify an object that should be non-mutable, as `const_cast` removes type-based write protection imposed on const objects. Union can also cause a type confusion problem when the attribute value which indicates the type of union is misused. A union data type is similar to a struct data type since it consists of a number of members with different names and types, each of which can be referred to individually. However, unlike the struct type, since union members all occupy the same location in memory, developers should use them mutually exclusively. The existence of these types therefore introduces the possibility of mistakenly referring to a member of a union that is invalid. For example, this problem can lead to an information leak [60].

Fuzzing for type violations Since HexType can identify type confusion issues at run-time, HexType can be utilized to find new type confusion vulnerabilities with the help of fuzzing frameworks such as AFL [59]. AFL is typically deployed with ASan [49], an LLVM-based sanitizer that checks for (partial) memory safety violations, to increase fuzzing throughput and precision. Without ASan, AFL detects only memory safety violations that result in a segmentation fault and cannot detect silent corruption. As AFL already supports ASan, integrating another sanitizer like HexType will be straight-forward, thereby extending AFL to trigger and detect dangerous type confusion vulnerabilities as well.

2.1.7 Related Work

In this section, we summarize previous research works on typecast verification. HexType focuses on type confusion attacks that violate pointer semantics in type-

casting operations. CaVer [41] first addressed such exploits due to type casting verification and identified eleven security vulnerabilities due to bad typecasting. Next, TypeSan [42] improved the performance and coverage over CaVer. Similar to HexType, both CaVer and TypeSan are implemented on top of the LLVM compiler framework, instrumenting code during compile time. For metadata allocation both CaVer and TypeSan use a disjoint metadata scheme. TypeSan uses a shadow memory scheme for metadata and CaVer implements a per thread red-black tree for stack objects and shadow memory for the heap. However, these schemes inhibit the identification of overall object allocation and increase the overhead. HexType uses a global, whole address space two layer object-to-type mapping scheme to reduce overhead and supports additional object allocation patterns through placement new and `reinterpret_cast`. Hence HexType vastly increases coverage compared to the aforementioned approaches. Performance is comparable despite the increased coverage.

UBSan [38], another typecast verification framework, works only for polymorphic classes. It relies on runtime type information (RTTI) and instruments only static_cast and checks the casting during runtime. Thus it can only handle polymorphic classes problem as well as requires manual source modification. This makes it difficult to use in large projects.

Several Control-Flow Integrity (CFI) techniques [43, 61–65] ensure the integrity by checking any invalid control-flow transfer within the program. However, these techniques address the type confusion problem only partially if control-flow hijacking is performed via type exploitation. Similarly, defenses [40, 66–68] that protect virtual calls from vtable hijacking attacks considers only the type of the virtual calls. These schemes do not address the overall bad casting problems. Another control-flow hijack mitigation technique is Code Pointer Integrity (CPI) [46, 69], which guarantees the integrity of all the code pointers in a program. This approach can prevent the accessibility of corrupted pointers, but does not block type casting attacks.

Bad type casting can lead to memory corruption attacks where an attacker can potentially get access to out-of-bounds memory of the cast object. Such attacks can be identified by existing mechanisms. Defense techniques focusing on memory corruption [11, 30, 32, 70–73] can detect exploits if a type confusion attack leads to memory access past the cast object. These techniques efficiently detect such attacks, but unlike HexType they cannot address type confusion issues.

2.2 Conclusion

Type casting vulnerabilities are a prominent attack vector that allows exploitation of large modern software written in C++. While allowing encapsulation and abstraction, object oriented programming as implemented in C++ does not enforce type safety. C++ oers several types of type casts and some are only checked statically and others not at all, at runtime an object of a dierent type can therefore incorrectly pass a type cast. To detect these illegal type casts, defenses need to both track the true allocated type of each object and replace all casts with an explicit check. HexType tracks the true type of each object by supporting various allocation patterns, several of which (such as placement new and `reinterpret_cast`) were not handled in previous work. While previous work focused only `static_casts`, HexType also covers `dynamic_cast` and `reinterpret_cast`. To limit the overhead of these online type checks, HexType both reduces the amount of incurred checks by removing checks that can be proven correct statically and limiting the overhead per check due to a set of optimizations. Our prototype results show that HexType has at least 1.1 – 6.1 times higher coverage on Firefox benchmarks. For SPEC CPU2006 benchmarks with overhead, we show a 2 – 33.4 times reduction in overhead. In addition, HexType discovered 4 new type confusion bugs in Qt and Apache Xerces-C++. The open-source version of HexType is available at https://github.com/HexHive/HexType.

3 V-TYPE: INLINE TYPE INFORMATION TO COUNTER TYPE CONFUSION

C++ is the world's most popular object-oriented programming language and is widely used in high-performance and large-scale applications. C++ enables high-level abstraction and modularity with low-level memory access and system intrinsics. This design choice of C++ comes at the price of safety. The programmer is responsible for carefully enforcing type and memory safety. In particular, the lack of type safety leads to type confusion vulnerabilities, which is one of the main attack vectors to compromise C++ applications. Type confusion vulnerabilities are logical errors caused by misuse of resources such as a pointer, object, or variable with an incompatible type, and may lead to unexpected program behavior. For example, through down-casting, a pointer of a descendent type can point to an object of a parent type. Using the illegally casted pointer, the attacker can trigger out-of-bound memory access by accessing the data fields that are not available in the parent type. Further, if there is important metadata such as a Vtable pointer next to the target object and can be accessed by a descendant type, an attacker can overwrite this information to hijack control flow. Exploitable type confusion vulnerabilities are found in a wide range of software products, such as Google Chrome V8 (CVE-2020-6418), WebKitGTK (CVE-2020-3897 and CVE-2020-3901), Autodesk FBX SDK (CVE-2020-7081), Adobe Flash Player (CVE-2020-3757), PhantomPDF (CVE-2020-15638), Firefox (CVE-2019-17026), Libxslt (CVE-2019-5815), Ghostscript (CVE-2018-19134), Adobe Flash Playeri (CVE-2018-4944), and Foxit PDF Reader (CVE-2018-9942). Due to deterministic runtime behaviors, these vulnerabilities can be exploitable by attackers.

In the past years, several type confusion detectors have been devised. These existing detectors can be categorized into two categories: (i) pointer casting monitor;

and (ii) pointer usage monitor. First, pointer casting monitors detect type confusion during casting. To detect type confusion, these approaches insert additional check instrumentation for all down-casting sites. At runtime, inserted check instrumentation invokes a runtime library function to verify type casting through checking the relationship between source and destination type of the casting operation. Clang CFI [37] uses the Vtable pointer to figure out source object type pointed by source pointer. However, this approach cannot verify typecasting between non-polymorphic types as only polymorphic objects have Vtable pointer. Disjoint metadata structure approaches such as Caver [74], TypeSan [3], and HexType [35] use additional disjoint metadata to maintain "object to type" mapping table. To maintain this disjoint metadata structure, these approaches insert additional instrumentation into all object allocation and deallocation sites. However, these approaches have high overhead issues because of the additional object tracking and metadata structure managing overhead. These approaches also have high false positive detection rate issues because of using outdated metadata created by incorrect object tracking (e.g., not removing deleted objects from corresponding metadata structure areas).

Second, pointer usage monitors [6, 75, 76] inspect load and store operations to detect type confusion when a pointer with the incorrect type gets referenced. These approaches mainly have high overhead issues as these have to insert all checks into deference sites although these show high detection coverage (i.e., checking other potentially dangerous ways such as union or `reinterpret_cast` through pointer usage monitoring).

These existing solutions have their own issues (e.g., high overhead or incorrect object tracking), and those issues ultimately make it more difficult to find type confusion bugs more effectively and accurately. Thus, we propose V-Type to improve the precision and performance of the type confusion detector. For this, V-Type uses inline type information available as the Vtable pointer in each runtime-allocated object. Considering this information only exists in polymorphic objects, we forcibly change non-polymorphic objects objects into polymorphic ones to ensure each object will

maintain type information. However, several difficult challenges (e.g., need to add additional instrumentation or to modify compiler) that arise when forcibly inserting type information into non-polymorphic objects are as following: (1) we need to insert additional constructor calls into all object allocation sites through the `malloc` family to initialize Vtable pointer, (2) we need to address standard violation issues, such as union having a polymorphic object as members, when we forcibly change the data layout, and (3) we need to address issues when the changed object will be used for communication with other components (e.g., libraries or kernel). To address these issues, V-Type inserts additional constructor calls to initialize Vtable pointer of forcibly changed polymorphic objects. Additionally, V-Type addresses standard violation issues by modifying the compiler without additional side effects. For communication issues with other components, V-Type rebuilds related libraries or software to make sure all components have the same type layout. However, considering the burden of rebuilding the kernel, we create wrapper functions to remove or reinsert Vtables, which are forcibly inserted, before and after system calls.

For optimization, V-Type only changes casting related non-polymorphic objects to polymorphic objects. Since objects unrelated to casting (distinguished via static analysis) will never be used for casting, V-Type does not need to change these non-polymorphic objects. To further increase detection coverage, V-Type also detects type confusion from `void*` or another unrelated type to the wrong dynamic type.

According to our evaluation, V-Type only incurs 0.81% overhead on SPEC CPU2006, which is 11 times faster than HexType. Additionally, V-Type shows 10 times higher detection coverage than Clang-CFI. Finally, compared to the previous approaches, V-Type has a low false positive rate for the type confusion detection because FuZZan is free from incorrect object tracking.

Our contributions are:

1. Designing and implementing V-Type and addressing three main issues when we forcibly change non-polymorphic type into polymorphic type.

2. Increasing detection coverage and reducing overhead using inline metadata structure while it shows low false positive rates.

3.1 Background

In this section, we summarize C++ facilities for subtype polymorphism and dynamic dispatch, their implications for memory layout and type confusion vulnerabilities.

3.1.1 Classes Hierarchies & Polymorphism

The C++ programming language facilitates subtype polymorphism as a mechanism for implementing generic functions. Whenever an instance (pointer or reference) of a type `T` is expected, this variant of polymorphism also accepts transitive subtypes of `T`. In particular, instances of derived classes can safely be treated as instances of their respective base classes. This allows reusing a single function for different derived classes and hence decreases code duplication. C++ implements access to member variables by laying out objects in memory such that instances of derived class start with the layout of their base class. Additional member variables of derived classes are appended afterwards.

Assume class `A` is subclassed by `B` and `C`. Objects of class `A` just have a single member variable at offset 0. Objects of derived classes `B` and `C` commence with the memory layout as `A` and proceed by appending their respective member variables. A function `f(A* a_obj)` specifying instances of class `A` as first parameter equally accept instances of class `B` and `C`. This is a characteristic property of subtype polymorphism which postpones determination of concrete type to runtime. As the runtime type of the object referenced by parameter 'obj' might vary, the compiler restricts member variable access to those variables common to all objects in this class hierarchy. I.e. only the member variables of class A are accessible through a variable of type A even if the referenced object's runtime type is B or C. This restriction imposed

```
class A {
   int a_int;
};

class B : A {
   int b_int;
};

class C : A {
   char* c_ptr;
};

void f(A* a_obj) {
  B* b_obj = static_cast<B*>(a_obj);
}
```

Figure 3.1.: Code example for type confusion

by the type system can be worked around by explicitly casting the variable of type A to a variable of type B or C. Due to postponing concrete types to runtime, compilers cannot determine whether the cast operation is correct in general. As compilers default to accepting the cast, it is the responsibility of the developer to ensure correct runtime types.

3.1.2 Type confusion

Consider the casting operation in line 14 of figure Figure 3.1. After the cast, member variables and methods of class B become accessible. Contrary to treating an object as instance of a base class, treating it as instance of a derived class has non-trivial implications w.r.t. correctness and security. Assuming `a_obj` indeed references an object of class B the program behaves correctly. However, a violation of this assumption results in incorrect program behavior. Should the runtime type of the object be class A, the member variables of B are outside of the allocation boundaries of the object. Hence accessing these variables causes an out-of-bound memory corruption.

Likewise, `a_obj` referencing an object of class C causes a memory corruption as well. Even though the access is not outside of the object's allocation boundaries, the member variables coincidentally sharing the same offset as '.' become overridden. In our example, the member variable `b_int` of B and `c_ptr` of C share the same offset of 8. Hence writing `b_int` corrupts the pointer stored in `c_ptr`.

3.1.3 C++ Casting

C++ features four different casting operators, each of them governed by distinct rules and properties. It is up to developers to choose the appropriate casting operator. The `const_cast` operator allows stripping the const and volatile property of an object. This operation is not without its own set of caveats, but unrelated to type confusion. Hence we consider incorrect `const_cast` to be out of scope.

The dynamic_cast operator allows to change the static type of an object while runtime verification validates the compatibility of the object's dynamic type. The validation routine mandated by the language specification generally requires the presence of metadata in form of runtime type information (RTTI). As RTTI is only available for polymorphic types dynamic_cast it cannot be used for runtime verification of casts to non-polymorphic objects. Embedding and traversing RTTI imposes significant overhead in terms of binary size and performance. Hence performance sensitive applications typically prohibit the usage of RTTI entirely. As a consequence, we cannot assume to have RTTI metadata available.

The static_cast operator changes the static type of an object similar to the dynamic_cast operator. However, no runtime checks are performed and RTTI is unnecessary. A limited amount of casting validation is performed solely at compile time. The verification checks whether there is a path of upcasts and downcasts allowing to move from source to target type. Lacking any runtime verification, downcasts with incorrect dynamic type pass the compile time checks. Casting a pointer from outside the class hierarchy (e.g. char* or a void*) into an object of a specific class is permissible as well. This eventuates in the same potential for incorrect casting, as there are no runtime checks. Incorrect usage of static_cast is the main culprit for type confusion in C++ applications.

The reinterpret_cast operator reinterprets the underlying bit pattern of the source type as target type. Verification happens neither at compile time nor at runtime. Ensuring the correctness of the cast is a manual task, similar to static_cast. As a consequence, the same security issues arise.

C-style casts are supported not only by C, but also C++. The compiler replaces this cast with one of three C++ casting operators. A const_cast is preferred over a static_cast, which in turn is more favorable than a reinterpret_cast. The replacement select the first operator nor causing a compilation error. Due to the definition of C-style casts in terms of C++ casting operators they inherit the underlying potential to type confusion vulnerabilities.

3.2 Design and Implementation

Figure 3.2 illustrates an overview of V-Type. V-Type is a Clang/LLVM-based sanitizer that checks for type confusion violations. Given the source code as input, V-Type forcibly changes any non-polymorphic type into a polymorphic type to ensure all allocated objects maintain type information through Vtable. Since there are no additional constructor calls when objects are allocated from `malloc` family, V-Type inserts additional constructor call instrumentation after `malloc` family. V-Type verifies the type casting operation based on Vtable pointer as Clang-CFI, although V-Type can still support non-polymorphic objects' type casting verification. For optimization, as non-polymorphic objects unrelated to type casting will not be used for type casting, V-Type only changes type related to casting. While compiling of a target program, including this additional process, V-Type generates a V-Type-hardened program. At runtime, V-Type verifies type casting and detects a type confusion error if there is illegal down-casting.

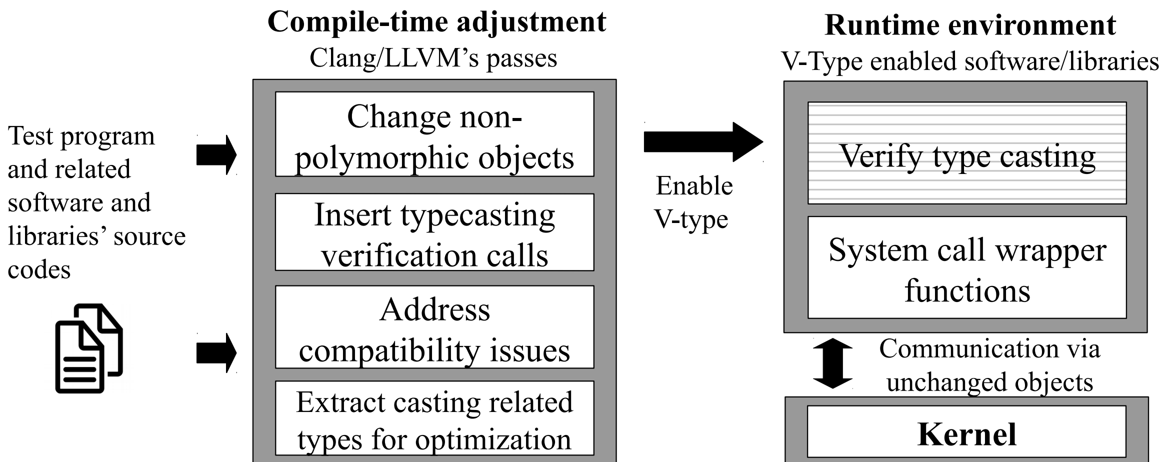


Figure 3.2.: Overview of V-Type's architecture and workflow.

3.2.1 Vtable inline metadata structure

For type casting verification, we must know the type information of the object pointed by the casting operation's source pointer. In order to know this, existing disjoint metadata structure approaches maintain object-to-type mapping metadata structure. However, this disjoint metadata structure has several limitations such as high overhead and false positive rate issues. To remove these limitations, V-Type uses Vtable inline metadata structure, which is Vtable pointer. Thus, to ensure non-polymorphic objects also have Vtable pointer, we change non-polymorphic class/structure into polymorphic class/structure as shown in Figure 3.3. For this, we modify Clang to insert Vtable pointer and create corresponding Vtable and constructors. For type casting verification, we rely on Clang CFI's [37] type casting verification function as it also uses Vtable pointer and has minor performance overhead for type casting verification.

3.2.2 Addressing compatibility issues

Forcibly changing non-polymorphic type into polymorphic type brings several challenges mainly because of the changed object layout. The main challenges are:

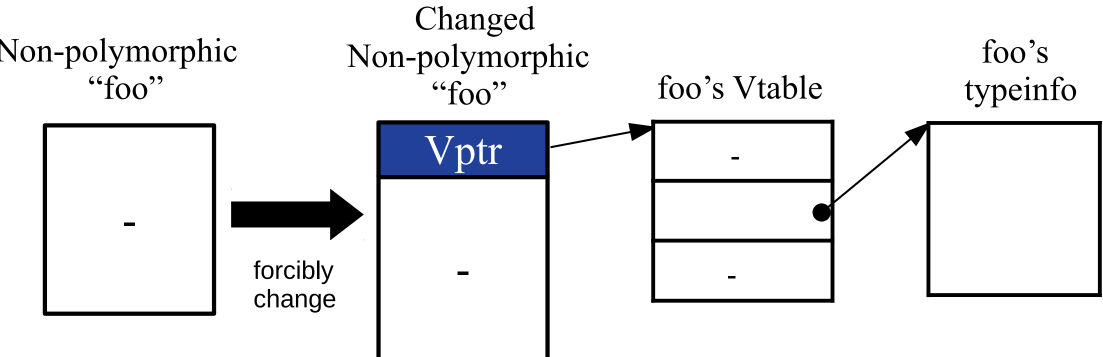


Figure 3.3.: Change non-polymorphic into polymorphic object.

(i) handling changed polymorphic objects allocated by the `malloc` family that is unaware of polymorphic types; and (ii) handling communication with other components using changed polymorphic objects; and (iii) addressing standard violation issues caused by the changed object layout. We address these issues by adding additional instrumentation to handle the allocation of changed objects by the `malloc` family, by rebuilding related components with FuZZan to handle compatibility issues with the communication between components, and by modifying the compiler to address standard violation issues.

Allocation with malloc family. Unlike `new` and `delete` operators in C++, although an object is created, `malloc` family (e.g., `malloc`, `realloc`, or `calloc`) does not call the constructor, which can initialize Vtable pointer. However, if the changed non-polymorphic objects allocated from this `malloc` family, it causes issues that allocated object's Vtable pointer is not initialized because of missed constructor call. To address this issue, if the `malloc` family functions allocate changed non-polymorphic object, we add an additional constructor call after target `malloc` family function to initialize Vtable pointer.

Communication with other libraries and software If a changed type in the target test application is used for communication with other components (e.g., passing a changed object as an external library function's parameter without the ability of the corresponding components, such as an external library, to recognize this change), this inconsistency will create issues. For instance, the Vtable pointer of changed objects can be misinterpreted as a predefined type by the external library functions. Thus, we rebuild other components with V-Type to make sure all components have the same type layout information. During the evaluation of SPEC CPU2006 [77], we find these issues when communicating with the C++ library, and therefore we rebuild the C++ library with V-Type in order to also change target non-polymorphic type into polymorphic type in the C++ library.

Communication with the operating system The interaction between the target test application and operating system through changed objects will create compatibility

```
union S {
    std::string str;
    std::vector<int> vec;
    ......
};

int main() {
    S s = {"HelloWorld"};
    .....
    // need to use "placement new" before switching
    // to a polymorphic object
    new (&s.vec) std::vector<int>;
    s.vec.push_back(10);
    ......
}
```

Figure 3.4.: Code example for adding placement_new before switching to union's a polymorphic object member.

issues just as library and software cases do. Considering the burden of rebuilding the kernel, we implement a wrapper function for system calls in libc to remove the Vtable pointer before sending changed objects to the kernel, and the wrapper function then adds the Vtable pointer back after the system call. To handle that, V-Type extracts the set of changed non-polymorphic and casting-related (for optimization) types, including inheritance (IS-A) or composition (HAS-A) relationships during compilation. Based on this set, V-Type 's system call wrappers can check whether the target parameter type has been modified. If the type of the variable has been modified, V-Type removes or adds a Vtable pointer from this object. Note that, V-Type also handles recursive object dependencies, i.e., if one of the reachable child objects' types have been modified, those must be adapted accordingly.

Standard violation issues Since V-Type forcibly changes non-polymorphic type into polymorphic type, this might violates the C++ language standard. For instance, before the C++11 standard (it is legal after C++11), unions cannot have polymorphic objects as member variables. Forcible changing a non-polymorphic type into a poly-

```
class T1 {
    const int t; // it cannot be defined in class
    public:
    T1(int c) : t(c) {}
    ......
};

// need to define the const variable outside the class
T1() : t( 100 ){}
```

Figure 3.5.: Code example for defining the const variable of class outside the class

morphic type may violate the standard. This is due to the inability to predict which member of the union will be activated to call its constructor. More specifically, let us suppose we have a union with two polymorphic classes and an `int` variable. When we create an object, we cannot predict how this object is constructed by default and what member's constructor should be called as we do not know which member of the union will be activated later. Thus, the union cannot have a polymorphic object as members or explicit constructors (e.g., `placement_new`) are generally needed when new members of the union are activated as shown in Figure 3.4. To automatically address this, V-Type checks each block and inserts constructor call instrumentation when a union's polymorphic class member is first activated in the target "block". In the same block, if a union is switched to another type (e.g, A → B → A), V-Type inserts an additional constructor call when a polymorphic class member is activated again.

Another issue is the initialization of `const` variables in changed polymorphic objects. More specifically, `const` variables cannot be not initialized during declaration [78]. Thus, to initialize the `const` member using a constructor, we have to use the initializer list that is used to initialize the data member of a class, as shown in Figure 3.5. However, it is hard to predict the `const` variable's initialization value when we forcibly insert constructors to initialize the target object's Vtable pointer. To

Table 3.1.: Detection coverage comparison with other pointer casting monitor approaches

Detector	**Polymorphic object**		**Non-Polymorphic object**	
	Based-to-derived	**void*/ unrelated type**	**Based-to-derived**	**void*/ unrelated type**
Caver	✓	✗	✓	✗
HexType	✓	✗	✓	✗
TypeSan	✓	✗	✓	✗
BiType	✓	✗	✓	✗
Clang-CFI	✓	✓	✗	✗
V-Type	✓	✓	✓	✓

address this issue, we modify the compiler to ignore this violation and then forcibly insert constructors without the initializer list only to initialize a Vtable pointer. This is because native constructors will initialize these const variables, even though Vtable's additionally inserted constructors skip the initialization of const variables.

3.2.3 Increase detection coverage

As shown in Table 3.1, the existing pointer use checking approaches detect two types of type confusion: (1) illegal down-casting during casts from a base class to a derived class and (2) illegal casts from a pointer of type void* or another unrelated type to the wrong dynamic type. No existing solution can cover both these two attacks because of overhead—we still need to check all casts including the static_cast—or because of a design issue (e.g., Clang-CFI only being able to verify the casting of polymorphic an objects). V-Type can cover both attack types as V-Type supports the casting of non-polymorphic objects and has no high overhead issue because V-Type is free from heavy object tracking and disjoint metadata structure management.

3.2.4 Only changing type related to type casting

Clearly, V-Type does not need to change non-polymorphic type into polymorphic type if target type is not used for type casting. This further helps to reduce overhead. For instance, V-Type does not need to add constructor calls for non-polymorphic objects unrelated to casting. As illustrated in Figure 3.6, V-Type performs the following two steps to classify type casting related non-polymorphic objects. First, V-Type checks all type casting and inserts type into the type set related to type casting if the target type is used for type casting operations. Next, the type in a casting operation can point to itself or any of the child type, which is always correct, as we assume that type confusions do not occur before bad casting. Thus, we extend a type casting-related type set to include all possible subtypes. Finally, when changing non-polymorphic type into polymorphic, V-Type selectively changes type only for type casting-related type. Note that this optimization is inspired by HexType's [35] only tracing type related to type casting optimization.

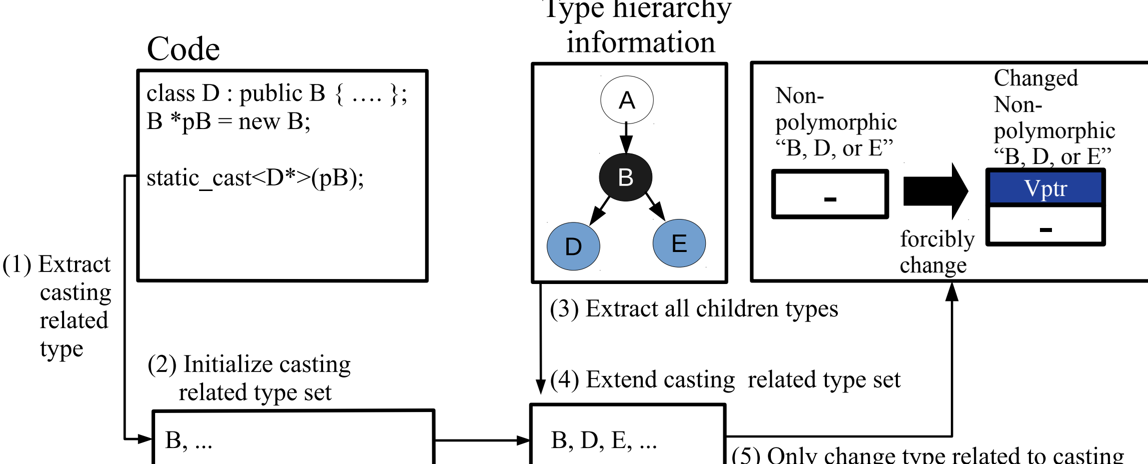


Figure 3.6.: An example of how V-Type creates a type set related to casting and only changes type using this set. In the example, we assume that the object of type B is typecast. V-Type will identify this type B and all its children types as casting related types.

3.2.5 Implementation

We have implemented V-Type based on the LLVM compiler infrastructure project (version 9.0.0). The V-Type implementation consists of 3.7k lines of code that we added to Clang, LLVM passes, and our compiler-rt runtime library. First, we modify Clang to (i) change non-polymorphic into polymorphic objects, and (ii) instrument target castings (Based-to-derived, `void*`, or unrelated type casting) that need to be verified. When we add the Vtable pointer into a class template, which provides a specification for generating classes based on parameters, we carefully modify compiler since Clang separetely handles class template objects, e.g, use `ClassTemplateDecl` (not `CXXRecordDecl`). We create LLVM passes to (i) address stability issues by, for example, inserting an additional constructor to initialize the Vtable pointer of changed non-polymorphic objects, and (ii) create type casting related type sets for optimization. More specifically, to address compatibility issues for allocation with the `malloc` family, V-Type recognizes the `malloc` family (i.e., `malloc`, `realloc`, or `calloc`) and its allocation size to carefully insert constructor calls. Finally, at runtime, V-Type 's runtime library functions (invoked by instrumentation for type casting operation) verify type casting operations using the Vtable pointer. Note that for type casting verification, we reuse the implementation of Clang-CFI (marked horizontal area in Figure 3.2) that extracting type information via Vtable pointer and comparing source and destination types' relationship (to verify type casting) because of its efficiency.

Additionally, to address communication issues with other components, we select LLVM libc++ [79] instead of libstdc++ [80] considering libc++'s compatibility with the LLVM compiler (our target compiler). We rebuild LLVM libc++ with V-Type to make sure class and structure in LLVM libc++ have the same type layout.

Table 3.2.: The evaluation of typecasting verification coverage against SPEC CPU2006. The # indicates the number of verified casting operations during SPEC CPU2006 test. The k represents thousand, m represents million, and b represents billion.

Benchmark	**Clang-CFI (#)**	**V-Type (#)**
Omnetpp	1,879m	2,520m
Xalancbmk	282m	283m
DealII	-	17b
Soplex	-	290k
Povray	-	-
Astar	-	-
Namd	-	-
Sum	2b	20b

3.3 Evaluation

In this section, we evaluate V-Type focusing on the following aspects: (i) the detection coverage; and (ii) runtime overhead;

Experimental Setting. All of our experiments are performed on a desktop running Ubuntu 19.10 LTS with a 32-core AMD Ryzen Threadripper 2990WX, 64GB of RAM, 1TB SSD.

Evaluation Target Programs. We have applied V-Type to all seven C++ benchmarks from SPEC CPU2006.

3.3.1 Coverage on type casting

Table 3.2 shows the type casting coverage of V-Type as compared to Clang-CFI, which is a similar Vtable pointer based type confusion detector. According to our evaluation, V-Type shows around 10 times more checks than Clang-CFI as V-Type supports non-polymorphic objects' type casting verification, which cannot be covered by Clang-CFI. Note that for this evaluation we only target down-casting operations to

Table 3.3.: SPEC CPU2006 benchmark performance overhead for Clang-CFI and V-Type. The first column with % denotes the ratio between Native and Clang-CFI. The next column denotes the ratio between Native and V-Type.

Benchmark	**Clang-CFI (%)**	**V-Type (%)**
Omnetpp	-0.60	0.00
Xalancbmk	1.99	2.12
DealII	0.60	1.45
Soplex	0.41	0.82
Povray	0.00	0.59
Astar	-0.59	-1.47
Namd	-0.40	-1.21
Sum	0.57	0.81

detect illegal down-casting just as other disjoint metadata structure approaches such as HexType do. However, V-Type will show better coverage than HexType when we detect type confusions from `void*` or another unrelated type to the wrong dynamic type.

3.3.2 Performance Overhead

Table 3.3 shows the performance overhead on the SPEC CPU2006. For all seven C++ benchmarks in SPEC CPU2006, V-Type shows significantly low overhead around 0.81%. Compared to Clang-CFI, V-Type just shows 0.24% slower performance than Clang-CFI although V-Type covers 10 times more type casting than Clang-CFI. V-Type also outperforms other disjoint metadata structure approaches such as HexType (8.54%) and Bitype (1.78%). More specifically, povray, astar, and namd, in SPEC CPU2006, do not perform any type casting operation. Thus, it shows almost zero overhead. Interestingly, remaining applications such as omnetpp, xalancbmk, dealII, and soplex also show low overhead (up to 2%) although they have considerable type casting operations (up to 17 billion). The main reason is that V-

Type removes heavy object tracking and disjoint metadata structure management overhead.

3.4 Future Work

Although V-Type's prototype demonstrates the efficiency of our inline type information-based type confusion detector, this evaluation result is limited to the SPEC CPU2006 benchmark. Therefore, we will further evaluate the V-Type project. In the future, we will assess V-Type's performance overhead, detection coverage, and memory usage with the more recent SPEC CPU2017 benchmark and complicated real-world applications such as Firefox and Chromium. We will also address additional compatibility issues, which can occur during the testing of these real-world applications. The evaluation of V-Type is not complete and we anticipate some engineering overhead when we implement support for large complex applications. We set this engineering efforts aside as future work.

4 FUZZAN: EFFICIENT SANITIZER METADARTA DESIGN FOR FUZZING

Fuzzing [81] is a powerful and widely used software security testing technique that uses randomly generated inputs to find bugs. Fuzzing has seen near ubiquitous adoption in industry, and has discovered countless bugs. For example, the state-of-the-art fuzzer American Fuzzy Lop (AFL) has discovered hundreds of bugs in widely-used software [59], while Google has found 16,000 bugs in Chrome and 11,000 bugs in over 160 other open source projects using fuzzing [82]. On its own, fuzzing only discovers a subset of all triggered bugs, e.g., failed assertions or memory errors causing segmentation faults. Bugs that silently corrupt the program's memory state, without causing a crash, are missed. To detect such bugs, fuzzers must be paired with sanitizers that enforce security policies at runtime by turning a silent corruption into a crash. To date, around 34 sanitizers [83] have been prototyped. So far, only the LLVM-based sanitizers ASan, MSan, LeakSan, UBSan, and TSan have seen wide-spread use. For brevity, we use *sanitizers* to refer to such frequently used sanitizers in the rest of the paper.

Unfortunately, sanitizers are designed for developer-driven software testing rather than fuzzing, and are consequently optimized for minimal per-check cost, not startup/teardown of the metadata structure. Consequently, they are based around a shadow-memory data structure wherein the address space is partitioned, and metadata is encoded into the "shadow" memory at a constant offset from program memory. Optimizing for long executions makes sense in the context of developer-driven software testing, which generally verifies correct behavior on expected input, leading to relatively long test execution times. Fuzzing has a more diverse set of inputs that cause both short (i.e., invalid inputs) and long running executions with billions of executions. For example, the Chrome developers use Address Sanitizer (ASan) for their unit tests and long-running integration tests [84]. However, the underlying design

decisions that make ASan a highly performant sanitizer for long running tests result in high performance overhead—up to 6.59×—for short executions, as observed in a fuzzing environment[1]. This high overhead reduces throughput, thereby preventing a fuzzer from finding bugs effectively.

We analyze the source of this overhead across a variety of sanitizers, and attribute the cost to heavy-weight metadata structures employed by these sanitizers. For example, Address Sanitizer maps an additional 20TB of memory for each execution, Memory Sanitizer (MSan) 72TB, and Thread Sanitizer (TSan) 97TB on a 64-bit platform. The high setup/teardown cost of heavy-weight metadata structures is amortized over the long execution of programs due to the low per-check cost. In contrast, a fuzzing campaign typically consists of massive amounts of short-lived executions, effectively transforming what is a large one-time cost into a large runtime cost. For example, Table 4.1 indicates that memory management is the main source of overhead for ASan under fuzzing on the Google fuzz test suite, accounting for 40.16% of the total execution time we observe. Memory management is the key bottleneck for using sanitizers with fuzzers, and has to date gone unaddressed.

Instead, increasing the efficiency and efficacy of fuzzing has received significant research attention on two fronts: (i) mechanisms that reduce the overhead of fuzzers [59, 85, 86]; and (ii) mechanisms that reduce the overhead of sanitization on longer running tests and conflicts between sanitizers [87–91]. These works address fuzzers and sanitizers in isolation, ignoring the core sanitizer design decision to optimize for long running test cases using a heavy-weight metadata structure that limits sanitizer performance in combination with fuzzers. Consequently, optimization of sanitizer memory management for short execution times remains an open challenge, motivated by the need to design sanitizers that are optimal under fuzz testing.

We present FuZZan, which uses a two-pronged approach to optimize sanitizers for short execution times, as seen under fuzzing: (i) two new light-weight metadata

[1]The average time for a single execution across the first 500,000 tests for the full Google fuzzer test suite is 0.61ms.

structures that trade significantly reduced startup/teardown costs [2] for moderately higher (or equivalent) per access costs and (ii) a dynamic metadata structure switching technique, which dynamically selects the optimal metadata structure during a fuzzing campaign based on the current execution profile of the program; i.e., how often the metadata is accessed. Each of our proposed metadata structures is optimized for different execution patterns; i.e., they have different costs for creating an entry when an object is allocated versus looking up information in the metadata table. By observing the metadata access and memory usage patterns at runtime, FuZZan dynamically switches to the best metadata structure *without* user interaction, and tunes this configuration throughout the fuzzing campaign.

We apply our ideas to ASan, which is the most widely used sanitizer [83, 92, 93]. ASan focuses on memory safety violations—arguably the most dangerous class of bugs, accounting for 70% of vulnerabilities at Microsoft [10]—and has already detected over 10,000 memory safety violations [7–9] in various applications (e.g., over 3,000 bugs in Chrome in 3 years [7]) and the Linux kernel (e.g., over 1,000 bugs [8,94]) by using a customized kernel address sanitizer (KASan). We further apply FuZZan to MSan and MOpt-AFL.

FuZZan improves fuzzing throughput over ASan by 52% when starting with empty seeds and 48% when starting with Google's seed corpus, averaged across all applications in the Google fuzzer test suite [95] as part of our input record/replay fuzzing experiment. Due to this improved throughput, FuZZan discovers 13% more unique paths (with an improvement in throughput of 61% compared to ASan) given the standard 24 hour fuzz testing with widely used real-world software and a provided corpus of starting seeds.

Crucially, FuZZan achieves this without *any* reduction in bug-finding ability. Therefore, FuZZan strictly increases the performance of ASan-enabled fuzzing, resulting in finding the *same* bugs in *less* time than using ASan with the same fuzzer.

Our contributions are:

---

[2] Compared to ASan, our min-shadow memory mode reduces the time that startup/teardown functions spend in the kernel by 62% on the first 500,000 tests across the full Google fuzzer test suite.

Table 4.1.: Comparison between native and ASan executions with a breakdown of time spent in memory management, and time spent for ASan's initialization and logging. Results are aggregated over 500,000 executions of the full Google fuzzer test suite [95]. Times are shown in milliseconds, and % denotes the ratio to total execution time.

Modes	**ASan's init time ms (%)**	**ASan's logging time ms (%)**	**Memory mgmt. time ms (%)**	**# page faults**
Native	0.00 (0.00%)	0.00 (0.00%)	0.05 (11.49%)	2,569
ASan	0.17 (10.58%)	0.30 (18.86%)	0.63 (40.16%)	11,967

1. Identifying and analyzing the primary source of overhead when sanitizers are used with fuzzing, and pinpointing the sanitizer design decisions that cause the overhead;

2. Designing and implementing a sanitizer optimization (FuZZan) and applying it to ASan; that is, we design several new metadata structures along with a dynamic metadata structure switching to choose the optimal structure at runtime. We also validate the generality of our design by further applying it to MSan and MOpt-AFL;

3. Evaluating FuZZan on the Google fuzzer test suite and other widely used real-world software and showing that FuZZan effectively improves fuzzing throughput (and therefore discovers more unique bugs or paths given the same amount of time).

4.1 Background and Analysis

We present an overview of fuzzing overhead and ASan (our target sanitizer). Further, we detail the design conflicts between ASan and fuzzing when used in combination.

4.1.1 Fuzzing overhead

Given the same input generation capabilities, a fuzzer's throughput (executions per second) is critical to its effectiveness in finding bugs. Greater throughput results in more code and data paths being explored, and thus potentially triggers more bugs. Running a fuzzer imposes some overhead on the program, a major component of which is the repeated execution of the target program's initialization routines. These routines—including program loading, `execve`, and initialization—do not change across test cases, and hence result in repeated and unnecessary startup costs. To reduce this overhead, many fuzzers leverage a *fork server*. A fork server loads and executes the target program to a fully-initialized state, and then clones this process to execute each test case. This ensures that the execution of each test case begins from an initialized state, and removes the overhead associated with the initial startup.

Another technique for reducing process initialization costs is *in-process fuzzing*, such as AFL's persistent mode and libFuzzer. In-process fuzzing wraps each test in one iteration of a loop in one process, thus avoiding starting a new process for each test. However, in-process fuzzing generally requires manual analysis and code changes [96, 97]. Additionally, in-process fuzzing requires the target code to be stateless across executions as all tests share one process environment, otherwise the execution of one test may affect subsequent ones, potentially leading to false positives. Consequently, testers should avoid in-process fuzzing for library code using global variables. Bugs found from in-process fuzzing may not be reproducible as it is not always possible to construct a valid calling context to trigger detected bugs in the target function, and side-effects across multiple function calls may not be captured [98]. Because of these limitations, in-process fuzzing is used on stateless functions in libraries, while the fork server model (i.e., out-of-process fuzzing) remains the most general fuzzing mode for fuzzing programs.

4.1.2 Address Sanitizer

All sanitizers leverage a customized metadata structure [83]. Out of many different metadata schemes, shadow memory (both direct-mapped or multi-level shadow) is the most widely used [3, 5, 12, 13, 75, 99–104]. ASan enforces memory safety by encoding the accessibility of each byte in shadow memory. Allocated (and therefore accessible) areas are marked and padded with inaccessible red zones. In particular, *direct-mapped shadow memory* encodes the validity of the entire virtual memory space, with every 8-bytes of memory mapping to 1-byte in shadow memory. Shadow memory encodes the state of application memory. The 8-bit value k encodes that the 8-k bytes of the mapped memory are accessible. The corresponding shadow memory address for a byte of memory is at:

$$addr_{shadow} = (addr >> 3) + offset$$

where *addr* is the accessed address. Generally, ASan only inserts redzones to the high address side of each object as the preceding object's redzone suffices for the low address side. ASan also instruments each runtime memory access to check if the accessed memory is in a red zone, and if so faults. ASan's effectiveness in detecting hard-to-catch memory bugs has led to its widespread adoption. It has become best practice [83] to use ASan (or KASan [105], the kernel equivalent) with a fuzzer to improve the bug detection capability.

4.1.3 Overhead Analysis of Fuzzing with ASan

To understand ASan's overhead with fuzzing, we analyze the Linux kernel functions used during fuzzing campaigns. Table 4.1 shows the overhead added by ASan, broken out across ASan's logging, ASan's initialization, and memory management. Our experiments measure the ratio of the time spent in the kernel functions compared to the total execution time for a number of target programs.

Note that memory management makes up 40.16% of ASan's total execution time, as opposed to 11.49% for the base case, and that memory management is more than double the overhead of ASan's logging and initialization *combined*. ASan's heavy use of the virtual address space results in 4.66× page faults compared to native execution. Our memory management overhead numbers reflect the time spent by the kernel in the four core page table management functions: (i) `unmap_vmas` (24.6%), (ii) `free_pgtable` (4.7%), (iii) `do_wp_page` (8.2%), and (iv) `sys_mmap` (2.6%).

Notably, `unmap_vmas` and `free_pgtable` correspond to 73% of ASan's measured memory management overhead across the four core page table management functions. The execution time for these two functions (`unmap_vmas` and `free_pgtable`) is 10x higher than when executing without ASan. To break this overhead down, when executing a test under the fork server mode, a fuzzer needs to create a new process for each test. During initialization, ASan reserves memory space (20TB total, including 16TB of shadow memory, and a separate 4TB for the heap on 64-bit platforms) and then poisons the shadow memory for globals and the heap. Accessing these pages incurs additional page faults, and thus page table management overhead in the kernel. Note that the large heap area causes sparse page table entries (PTEs), which increase the number of pages used for the page table and memory management overhead.

Existing techniques to deal efficiently with large allocations do not help here. Lazy page allocation of the large virtual memory area used by ASan does not mitigate memory management overhead in this case, as many of the pages are accessed when shadow memory is poisoned. Poisoning forces a copy even for copy-on-write pages, and thus increases page table management cost. During execution, memory allocations and accesses cause additional shadow memory pages to be used, again with page faults and page table management. When the process exits, the kernel clears all page table entries through `unmap_vmas` and releases memory for the page table (via `free_pgtables`). The cost of these two functions are correlated with the number of physical pages used by the process. As fuzzing leads to repeated, short executions, such bookkeeping introduces considerable memory management overhead.

In contrast to these active memory management functions, `sys_mmap` only accounts for 7% memory management overhead of ASan. This is the expense for reserving all virtual memory areas. However, large areas that are actively accessed by ASan incur considerable additional expenses as detailed above.

For completeness, we note that our analysis finds that ASan performs excessive "always-on" logging (18.86%) by default, and that ASan's initial poisoning of global variables (10.58%) is inefficient. Combined, these additional sources of overhead account for 29.44% overhead. We address these engineering shortcomings in our evaluation, but they are neither our core contributions nor the choke point in fuzzing with ASan.

4.2 FuZZan design

FuZZan has two design goals: (1) define new light-weight metadata structures, and (2) automatically switch between metadata structures depending on the runtime execution profile. In this section, we present how we design each component of FuZZan to achieve both goals, as illustrated in Figure 4.1.

4.2.1 FuZZan Metadata Structures

To minimize startup/teardown costs while maintaining reasonable access costs, FuZZan introduces two new metadata structures: (i) a Red Black tree (RB tree) metadata structure, which has low startup and teardown costs, but has high per-access costs; and (ii) min-shadow memory, which has medium startup/teardown costs and low per-access costs (on par with ASan). Table 4.2 shows a qualitative comparison of the different metadata schemes that we propose in this section, see Table 4.4 for quantitative results. The RB tree is optimal for short executions with few metadata accesses as it emphasizes low startup and teardown costs, while min-shadow memory is best suited for executions with a mid-to-high number of metadata accesses as it

Table 4.2.: Comparison of metadata structures.

Metadata Structures		Startup/ Teardown Cost	Access Cost
ASan shadow memory		High	Low
FuZZan	Customized RB-tree	Low	High
	Min-shadow memory	Medium	Low

has lower per metadata access costs while still avoiding the full startup/teardown overhead imposed by ASan's shadow memory.

Customized RB-Tree

To optimize ASan's metadata structure for test cases where a fuzz testing application only executes for a very short time with few metadata accesses, we introduce a customized RB tree, shown in Figure 4.2. Nodes in the RB tree store the redzone for

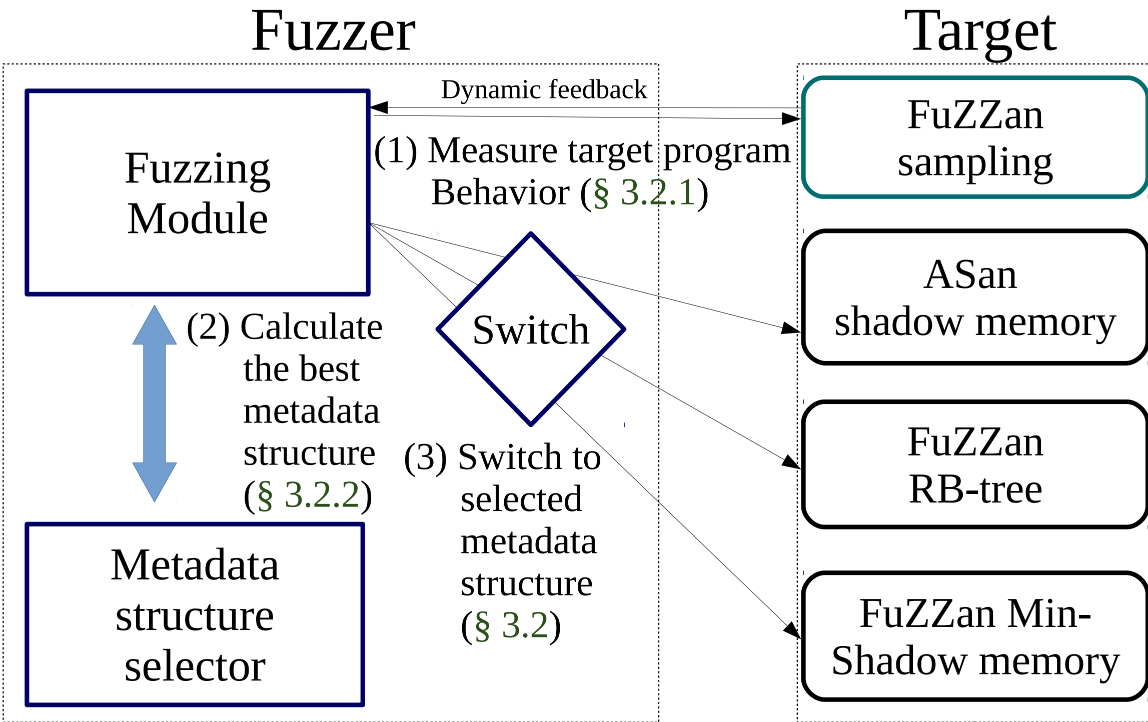


Figure 4.1.: Overview of FuZZan's architecture and workflow.

each object. Although each metadata access operation (insert, delete, and search) in the RB tree is slower than its counterpart in the shadow memory metadata structure, our RB tree has the following benefits: (i) low total memory overhead (leading to low startup/teardown overhead); (ii) removal of poisoning/un-poisoning page faults (as each RB tree node compactly stores the redzone addresses and these nodes are grouped together in memory); and (iii) a faster range search than shadow memory for operations such as `memcpy`. For example, in order to check `memcpy`, ASan must validate each byte individually using shadow memory. However, in our approach, we can verify such operations through only two range queries for `memcpy`'s source and destination memory address range.

In our RB tree design, when an object is allocated (e.g., through `malloc`), the range of the object's high address redzone is stored in a node of the RB tree. During a query, if the address range of the target is lower than the start address of the node, we search the left subtree (and vice versa). If the address is not found in the tree, it is a safe memory access. During redzone removal, the requested address range may

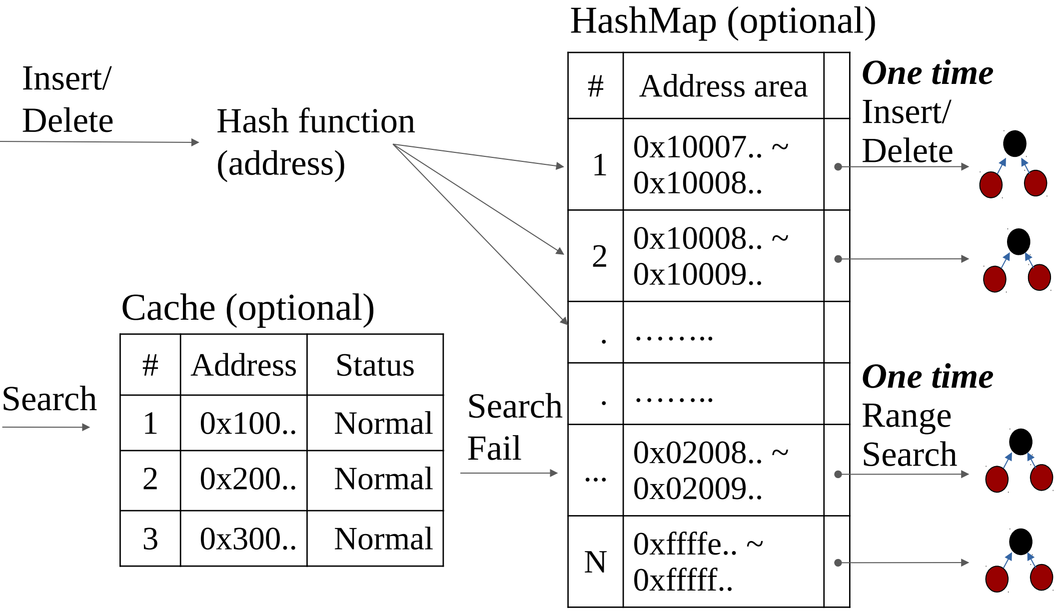


Figure 4.2.: Design of FuZZan's customized RB-tree.

only be a subset of an existing node's range (and not the full range of a target node in the RB tree). In this case, the RB tree deletes the existing RB tree node, creates new RB tree nodes which have non-overlapping address ranges (e.g., the left and right side of an overlapped area), and inserts these nodes into the RB tree. Since we reuse ASan's memory allocator and memory layout (e.g., redzones between objects and a quarantine zone for freed objects), FuZZan provides the same detection capability as ASan.

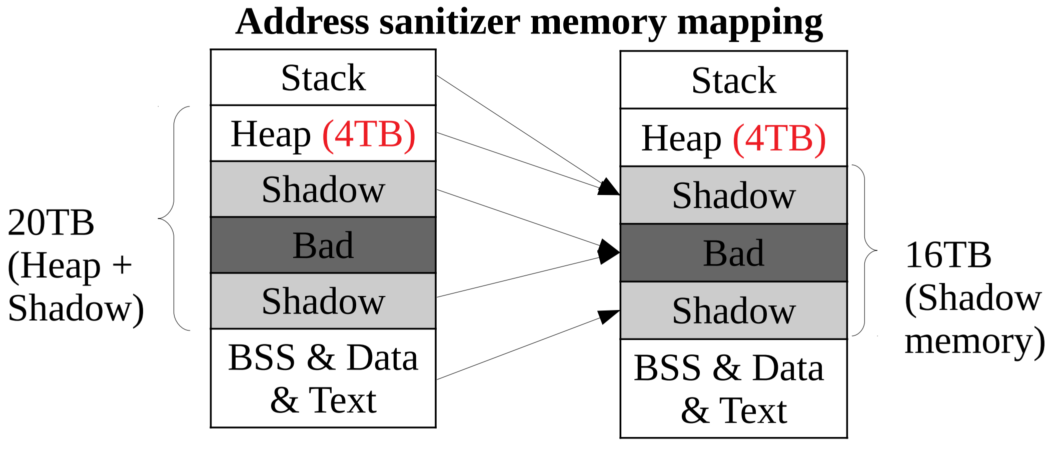
Min-shadow memory

The idea behind Min-shadow memory (for executions with a mid-to-high number of metadata accesses) is to limit the accessible virtual address space, effectively shrinking the size of the required shadow memory. As the size of shadow memory is a key driver of overhead in the fuzzing environment, this enhances performance.

Figure 4.3 illustrates how min-shadow memory converts a 64-bit program running in a 48-bit address space to run in a 32-bit address space window (1GB for the stack, 1GB for the heap, and 2GB for the BSS, data, and text sections combined). Note that pointers remain 64 bits wide and the code remains unchanged: the mapped address space is simply restricted, allowing min-shadow memory to have a partial shadow memory map. To shrink a program's memory space, we move the heap (by modifying ASan's heap allocator) and remap the stack to a new address space. Min-shadow memory remaps parts of the address space but programs remain 64-bit programs. To accommodate larger heap sizes, we create additional min-shadow memory binaries with heap sizes of 4GB, 8GB, and 16GB.

Our approach allows testing 64-bit code with 64-bit pointers without having to map shadow tables for the entire address space. We disagree with the recommendation of the ASan developers to compile programs as 32-bit executables, as changing the target architecture, pointer length, and data type sizes will hide bugs. Furthermore, min-shadow memory provides greater flexibility compared to using the `x32`

ABI [106] mode (i.e., running the processor in 64-bit mode but using 32-bit pointers and arithmetic, limiting the program to a virtual address space of 4GB), as min-shadow memory can provide various heap size options.



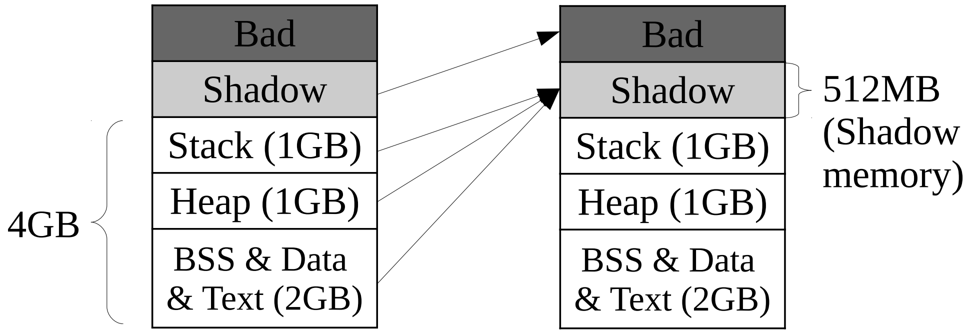


Figure 4.3.: ASan and min-shadow memory modes' memory mapping on 64-bit platforms. ASan (top) reserves 20TB memory space for heap and shadow memory, conversely, min-shadow memory mode (bottom) reserves 4512MB memory space for heap and shadow memory. Each application's stack, heap, and other sections (BSS, data, and text) map to the corresponding shadow regions. Further, the shadow memory region is mapped inaccessible.

4.2.2 Dynamic metadata structure switching

Dynamic metadata structure switching automatically selects the optimal metadata scheme based on observed behavior. At the beginning of a fuzzing campaign, dynamic metadata structure switching assesses the initial behavior and then periodically samples behavior, adjusting the metadata structure if necessary. Our intuition for dynamic metadata structure switching is that, during fuzzing, metadata access patterns and memory usage remain similar across runs and change in phases. While the fuzzer is mutating a specific input, the executions of the newly created inputs are *similar regarding their control flow and memory access patterns* compared to the source input. However, new coverage may lead to different execution behaviors. We therefore design a *dynamic metadata structure switching* technique that periodically and conditionally samples the execution and adjusts the underlying metadata structure according to the observed execution behavior.

Dynamic metadata structure switching compiles the program in four different ways in preparation for fuzzing: ASan, RB tree, min-shadow memory, and sampling mode. The sampling mode repeatedly samples the runtime parameters and then selects the optimal metadata structure. The selection of the optimal metadata structure is governed by FuZZan's metadata structure switching policy.

Sampling mode

The sampling mode measures the behavior of the target program using the min-shadow memory-1GB metadata mode and, based on the behavior, reports the currently optimal metadata structure. The sampling mode profiles the following parameters: (i) the number of metadata accesses during insert, delete, and search; and (ii) memory consumption. Note that this information can be collected by simple counters: profiling is therefore light-weight.

Dynamic metadata structure switching starts in sampling mode and selects the optimal mode based on the observed behavior. Dynamic metadata structure switching

then periodically (e.g., every 1,000 executions) and conditionally (e.g., when the fuzzer starts mutating a new test case) samples executions to select the optimal metadata structure based on the current behavior. To reduce the cost of periodic sampling, dynamic metadata structure switching implements a continuous back-off strategy that gradually increases the sampling interval as long as the metadata structure does not change (similar to TCP's slow-start [107]). Note that bugs may be triggered during sampling mode. As such, we maintain ASan's error detection capabilities while sampling to ensure that we do not miss any bugs.

Metadata structure switching policies

Our metadata structure switching policy is based on a mapping of metadata access frequency to the corresponding metadata structure. This heuristic is relatively simple in order to achieve a low sampling overhead. To determine the best cutoff points, we compile all 26 applications in Google's fuzzer test suite in two different ways: RB tree and min-shadow memory. We then test these different configurations against 50,000 recorded inputs and determine the best metadata structure depending on the observed parameters, measuring execution time. Profiling reveals that the frequency of metadata access (insert, delete, and search) is the primary factor that influences metadata structure overhead, which confirms our original assumption. In this policy, depending on the metadata access frequency, we select different metadata structures (based on statistics from profiling): RB tree if there are fewer than 1,000 accesses; and min-shadow memory if there are more than 1,000 accesses. Additionally, if the selected heap size goes beyond a threshold, we sequentially switch to other modes (min-shadow memory-4G, 8G, 16G, and ASan), thus increasing heap memory for continuous fuzzing.

4.3 Implementation

We implement FuZZan's two metadata structures and dynamic metadata structure switching mode on top of ASan in LLVM [53] (version 7.0.0). We support and interact with AFL [59] (version 2.52b). To address the other sources of overhead in ASan (shown in Table 4.1), we also implement two additional optimizations: (i) removal of unnecessary initialization; and (ii) removal of unnecessary logging. Our implementation consists of 3.5k LOC in total (mostly in LLVM, with minor extensions to AFL).

RB-tree. The RB tree requires modifications to ASan's memory access instrumentation, as our RB tree is not based on a shadow memory metadata structure. Thus, we modify all memory access checks, including interceptors, to use the appropriate RB tree operations instead of the equivalent shadow memory operations. As an optimization, and for compatibility with min-shadow memory mode, the RB tree mode also reserves 1GB for the heap memory allocator. A compact heap reduces memory management overhead. The RB tree mode is used when fuzz tests only execute for a very short time with few metadata accesses (i.e., they allocate relatively a small amount of memory).

Min-shadow memory. Unlike the RB tree, we are able to repurpose ASan's existing memory access checks, as the min-shadow memory metadata structure is based on a shadow memory scheme. To shrink a 64-bit program's address space, we modify ASan's internal heap setup and remap the stack using Kroes et al.'s linker/loader tricks [108]. More specifically, based on this script, we hook `__libc_start_main` using "`LD_PRELOAD`" and then `remap` the stack to a new address, update `rbp` and `rsp`, and then call the original `__libc_start_main`. This allows us to reduce ASan's shadow map requirements from 16TB of mapped (but not necessarily allocated) virtual memory to 512MB (1 bit of shadow for each byte in our 4GB address space window). We also create an additional 192MB shadow memory for ASan's secondary allocator and dynamic libraries (which are remapped above the stack). Fi-

nally, we implement four different min-shadow memory modes with increasing heap sizes (1GB, 4GB, 8GB, and 16GB) to handle the different memory requirements of a variety of programs.

Heap size triggers. As previously stated, min-shadow memory is configured for different heap sizes. We therefore use out of memory (OOM) errors to trigger callbacks that notify FuZZan to increase the heap size.

AFL modifications. The target program is compiled once per FuZZan mode. By default, AFL uses a random number generator (RNG) to assign an ID to each basic block within the target program. Unfortunately, this would result in the same input producing different coverage maps across the set of compiled targets, breaking AFL's code coverage analysis. We therefore modify AFL to use the same RNG seed across the set of compiled targets. This ensures that the same input produces the same coverage map across all compiled variants.

Removing unnecessary initialization. ASan makes a number of global constructor calls on program startup, performing several `do_wp_page` calls for copy-on-write. These constructor calls are unnecessarily repeated each time AFL executes a new test input, leading to redundant operations. Unfortunately, the AFL fork server is unaware of ASan's initialization routines. Therefore, to remove unnecessary (re-)initialization across fuzzing runs, we modify ASan's LLVM pass so that global variable initialization occurs *before* AFL's fork server starts. This is achieved by adjusting the priority of global constructors which contain ASan's initialization function.

Removing unnecessary logging. ASan provides logging functionality for error reporting (e.g., saving allocation sizes and thread IDs during object allocation). Unfortunately, this logging functionality introduces additional page faults and performance overhead. However, this logging is unnecessary because fuzzing inherently enables replay by storing test inputs that trigger new behavior. Complete logging information can be recovered by replaying a given input with a fully-instrumented program. We therefore identify and disable ASan's logging functionality (e.g., `StackDepot`) for fuzzing runs, allowing it to be reenabled for reportable runs.

4.4 Evaluation

We provide a security and performance evaluation of FuZZan. First, we verify that FuZZan and ASan have the same error-detection capabilities. Second, we evaluate the efficiency of FuZZan's new metadata structures and dynamic metadata structure switching mode using deterministic input from a record/replay infrastructure to ensure fair comparisons. Next, to consider the random nature of fuzzing and to show FuZZan's real-world impact, we evaluate FuZZan's efficiency without deterministic input. Here we evaluate the number of code paths found by FuZZan in a 24 hour time period, demonstrating the impact of FuZZan's increased performance. We also measure FuZZan's bug finding speed by using known bugs in Google's fuzzer test suite to verify that FuZZan maximizes fuzzing execution speed while providing the exact same bug detection capabilities as ASan. Finally, we port FuZZan to another sanitizer (MSan) [99] and another AFL-based fuzzer (MOpt-AFL) [109] to verify its flexibility.

Evaluation setup. All of our experiments are performed on a desktop running Ubuntu 18.04.3 LTS with a 32-core AMD Ryzen Threadripper 2990WX, 64GB of RAM, 1TB SSD, and Simultaneous MultiThreading (SMT) disabled (to guarantee a single fuzzing instance is assigned to each physical core). Across all experiments, we apply FuZZan to AFL's fork server mode, which is a widely-used and highly optimized out-of-process fuzzing mode. We evaluate FuZZan on all applications in the Google fuzzer test suite [95] and other widely used real-world software.

Evaluation strategy. Evaluating fuzzing effectiveness is challenging. In a recent study of how to evaluate fuzzing by Klees et. al. [110], the authors find that the inherent randomness of the fuzzer's input generation can lead to seemingly large but spurious differences in fuzzing effectiveness. However, we are at an advantage as we do not need to compare different fuzzers nor do we change the input generation. We therefore record the fuzzer-generated inputs during a regular run of AFL, and then replay these recorded inputs to compare our different ASan optimizations to the same baseline,

Table 4.3.: Three different metadata structure modes' detection capability based on the Juliet Test Suite for memory corruption CWEs. FuZZan and ASan have identical results. Good tests have no memory corruption to check for false positives. Bad tests are intentionally buggy to check for false negatives.

CWD (ID)	**Good tests (Pass/Total)**	**Bad tests (Pass/Total)**
Stack-based Buffer Overflow (121)	2,432 / 2,432	2,314 / 2,432
Heap-based Buffer Overflow (122)	1,594 / 1,594	1,328 / 1,594
Buffer Under-write (124)	682 / 682	641 / 682
Buffer Over-read (126)	524 / 524	359 / 524
Buffer Under-read (127)	682 / 682	641 / 682
Total	5,914 / 5,914	5,283 / 5,914

effectively controlling for randomness in input generation by using the same input for all experiments. For our experiments we record the first 500,000 executions for replay, yielding a large enough test corpus for reasonable performance comparisons. We also undertake a real-world fuzzing campaign (i.e., without inhibiting fuzzing randomness by record/replay) to measure FuZZan's real-world impact on code path exploration. Finally, Klees et. al. demonstrate the importance of the initial seed(s) when evaluating fuzz testing, as performance can vary substantially depending on what seed is used. We therefore compare two scenarios: (i) starting with the empty seed; and (ii) starting with a set of valid seeds (we use Google's provided seeds for the input record/replay experiment and randomly selected seeds of the right file type for our real-world fuzz testing).

4.4.1 Detection capability

We verify that FuZZan and ASan detect the same set of bugs in three different ways. First, we use the NIST Juliet test suite [111], which is a collection of test cases containing common vulnerabilities based on Common Weakness Enumeration (CWE). We use the full Juliet test suite for memory corruption CWEs to verify

FuZZan's capability to detect the same classes of bugs as ASan, without introducing false positives or negatives. Second, to verify that FuZZan and ASan also have the same detection capability under fuzz testing, we use the Google fuzzer test suite and our recorded input corpus. Finally, we leverage the complete set of ASan's public unit tests as a further sanity check.

For the Juliet test suite (Table 4.3), we select CWEs related to memory corruption bugs and obtain the same detection results from the three different modes (ASan's shadow memory, RB tree, and min-shadow memory). To validate FuZZan against ASan on the Google fuzzer test suite, we compare AFL crash reports across the full set of target programs in the Google fuzzer test suite with our recorded inputs (to identify both false positives and false negatives). Note that we force ASan to crash (the default setting under fuzz testing) when a memory error happens as fuzzers depend on program crashes to detect bugs. As expected, FuZZan's different modes all obtain the same crash results as ASan. However, we encounter minor differences between FuZZan and ASan when sanity-checking on the ASan unit tests. These differences are due to internal changes we made when developing FuZZan, such as min-shadow memory's changed memory layout (failed test cases include features such as fixed memory addresses).

4.4.2 Efficiency of new metadata structures

We perform input record/replay fuzz testing to evaluate the effectiveness of FuZZan's new metadata structures. Doing so isolates the effects of our metadata structures by removing most of the randomness/variation from a typical fuzzing run.

Over the full Google fuzzer test suite, the RB tree, without any other optimization, shows shorter execution times than ASan if the target application has less than 1,000 metadata accesses; conversely, the RB tree is slower than ASan when the target application has more than 1,000 metadata accesses. On average, as shown in

Table 4.4, several applications in the Google fuzzer test suite have more than 1,000 metadata accesses, and so RB tree is overall slower than ASan on average.

Despite being slower on average, the RB tree can be faster on individual applications and inputs. For instance, FuZZan in RB tree mode demonstrates a 19% performance improvement (up to 45% faster) for 15 applications (the remaining 11 applications show higher overhead compared to ASan) when benchmarked using the inputs generated from an empty seed. On the subset of applications for which seeds are provided, RB tree shows less performance improvement (17% and up to 39% faster) for 14 applications (the remaining 12 applications show higher overhead than ASan) when benchmarked using inputs generated from those seeds as provided seeds help to create valid input, lengthening execution times and thus metadata accesses. Note that RB tree shows the best fuzzing performance when the target application (e.g., `c-ares`) has less 1,000 metadata access. Additionally, even for applications where RB tree is slower across all inputs, it is still *faster* on inputs with few metadata accesses. The variable performance of RB tree, which is highly dependent on the number of metadata accesses, highlights the need for dynamic metadata structure switching to automatically select the optimal metadata structure.

Min-shadow memory mode, without additional optimization, outperforms ASan on all 26 programs (for both empty and provided seeds), as shown in Table 4.4. More specifically, the average improvement is 45% when starting with an empty seed and 43% when starting with the provided seeds. While different min-shadow memory heap configurations show gradual increases in memory overhead (from 1GB to 16GB, in line with the heap size), all of them outperform ASan (at worst, min-shadow memory is still 36% faster than ASan with a provided seed).

Additionally, both metadata configurations can utilize our two engineering optimizations; i.e., removing logging and modifying ASan's initialization (as described in section 4.3). Table 4.5 shows that the average improvement of removing unnecessary logging is 24% when starting with an empty seed and 19% when starting with the provided seeds. Similarly, modifying the initialization sequence improves performance

Table 4.4.: Comparison between four min-shadow memory modes, RB tree, Native, and ASan execution overhead during input record and replay fuzz testing with empty and provided seed sets. The time (s) indicates the average of all 26 applications' execution time during testing. Positive percentage (e.g., 20%) denotes overhead while negative percentage indicates a speedup.

Modes	**Empty seed**			**Provided seed**		
	time (s)	**vs. Native (%)**	**vs. ASan (%)**	**time (s)**	**vs. Native (%)**	**vs. ASan (%)**
Native	199	-	-	274	-	-
ASan	809	306	-	1,105	303	-
RB tree	1,541	673	90	3,308	1,106	199
Min-1G	443	122	-45	632	131	-43
Min-4G	465	133	-43	666	143	-40
Min-8G	467	134	-42	685	150	-38
Min-16G	477	139	-41	710	159	-36

Table 4.5.: Comparison between FuZZan's three different optimization modes, native min-shadow memory (1G) mode, and min-shadow memory (1G) mode with FuZZan's two optimizations, and dynamic metadata structure switching (Dynamic) mode execution overhead during all 26 applications' input record and replay fuzz testing.

Modes	**Empty seed**			**Provided seed**		
	time (s)	**vs. Native (%)**	**vs. ASan (%)**	**time (s)**	**vs. Native (%)**	**vs. ASan (%)**
Logging-Opt.	613	208	-24	891	225	-19
Init-Opt.	686	244	-15	987	260	-11
Logging+Init	552	177	-32	826	201	-25
Min-Shadow	443	122	-45	632	131	-43
Min-Shadow-Opt.	385	93	-52	574	109	-48
Dynamic	387	94	-52	578	111	-48

Table 4.6.: Comparison between native, ASan, min-shadow memory (1G), two optimizations with min-shadow memory executions with a breakdown of time spent in memory management, and time spent for ASan's initialization and logging. Results are aggregated over 500,000 executions of the full Google fuzzer test suite. Times are shown in milliseconds, and % denotes the ratio between single execution time and each section execution's time.

Modes	**ASan's init time ms (%)**	**ASan's logging time ms (%)**	**Memory manage time ms (%)**	**Page fault #**
Native	0.00 (0.00%)	0.00 (0.00%)	0.05 (11.49%)	2,569
ASan	0.17 (10.58%)	0.30 (18.86%)	0.63 (40.16%)	11,967
Min	0.10 (9.51%)	0.01 (1.33%)	0.24 (24.77%)	7,386
Min-Opt.	0.00 (0.00%)	0.00 (0.00%)	0.24 (24.71%)	6,139

by 15% when starting with an empty seed and by 11% when starting with the provided seeds. Combining the two engineering optimizations with min-shadow memory demonstrates synergistic effects: the combined performance is 52% (7% better than native min-shadow memory) faster for empty seeds, and 48% (5% better than native min-shadow memory) faster for provided seeds.

Overall, FuZZan's metadata structures show better performance than ASan's shadow memory for all 26 Google fuzzer test suite applications. As shown in Table 4.6, the main reasons for FuZZan's improvement are: (i) the smaller memory space reduces memory management overhead as page table management is more lightweight and incurs fewer page faults, (ii) our two engineering optimizations further reduce overhead and number of page faults by removing unnecessary operations, and (iii) the min-shadow memory mode has the same $O(1)$ time complexity for accessing target shadow memory as accessing the original ASan metadata. However, we also observe that the RB tree is faster than min-shadow memory for some configurations and programs (e.g., `c-ares-CVE`). This motivates the need for dynamic metadata structure switching, which observes program behavior and dynamically selects the best metadata structure based on this behavior.

4.4.3 Efficiency of dynamic metadata structure

As described in subsection 4.2.2, the dynamic metadata structure switching mode leverages runtime feedback to select the optimal metadata structure, dynamically tuning fuzzing performance according to runtime feedback. The intuition behind the dynamic metadata structure switching mode is that (i) no single metadata structure is best across all applications, (ii) the best metadata structure is not known a priori, so the analyst cannot pre-select the optimal metadata structure, and (iii) fuzzing goes through phases, e.g., alternating between longer running tests (e.g., exploring new coverage) and shorter running tests (e.g., invalid input mutations searching for new code paths). A consequence of the phases of fuzzing is that the same metadata structure is not optimal for every input to a given application. To verify the effectiveness of dynamic metadata structure switching, which is implemented based on these intuitions, we apply dynamic metadata structure switching mode to fuzz testing for seven widely used applications for fuzzing and all 26 applications' in Google's fuzzer test suite.

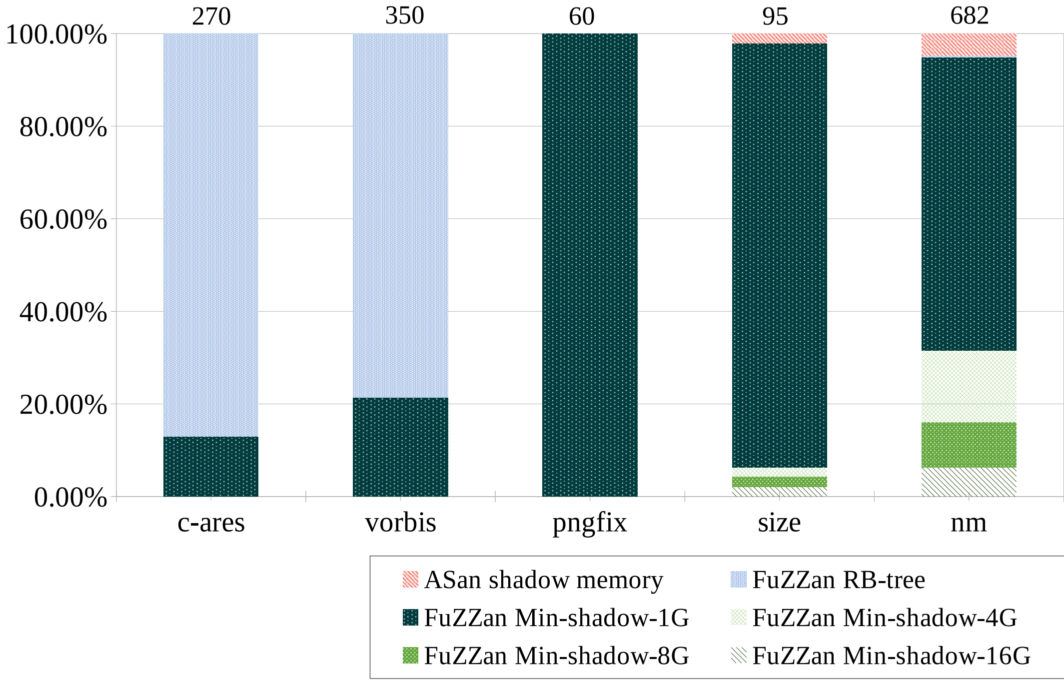


Figure 4.4.: Evaluating the frequency of metadata structure switching and each metadata structure selection over the first 500,000 tests each for c-ares and vorbis in Google's fuzzer test suite and pngfix, size, and nm. The number on each bar indicates the total metadata switches.

Our evaluation of dynamic metadata structure switching validates our intuitions, as shown in Figure 4.4. Observe that different applications are dominated by different metadata structures, e.g., `c-ares` for RB tree and `pngfix` for min-shadow memory. This is because dynamic metadata structure switching automatically selects the optimal metadata structure (which is unknown a priori). Because dynamic metadata structure switching is automatic, it prevents users from making errors such as selecting RB tree for applications with a large number of metadata accesses, and removes the need for any user-driven profiling to make metadata decisions. Further, dynamic metadata structure switching scales alongside with the required memory of applications as it increases when the fuzzer finds deeper test cases, as evidenced by `size`, `pngfix`, or `nm` switching to different min-shadow memory modes (4GB, 8GB, and 16GB heap sizes), without user intervention. Without dynamic metadata structure switching, inefficient min-shadow memory modes would be used at the beginning of

fuzzing campaigns, or users would have to pause and restart fuzzing campaigns to change metadata modes.

As an extreme example highlighting the need for automatic metadata switching, the `nm` benchmark changes metadata structures 682 times, underscoring the infeasibility of having a human analyst determine the single best metadata structure.

As a result of these factors, FuZZan's dynamic metadata structure switching mode improves performance over ASan by 52% when starting with empty seeds and 48% when starting with non-empty seeds. Further, ASan has 306% and FuZZan has 94% (212% less) overhead with empty seeds and ASan has 303% and FuZZan has 111% (192% less) overhead with non-empty seeds compared to native execution. Note that dynamic metadata structure switching has identical fuzzing performance to using min-shadow memory with 1GB heap alone, and improves performance over RB tree up to 870%. Consequently, automating metadata selection is not adding noticeable overhead, while substantially improving user experience. We recommend using dynamic metadata structure switching mode for the following four reasons: (i) if the target application exceeds FuZZan's heap memory limit (1GB), dynamic metadata structure switching automatically increases the heap size for the few executions that require it (a fixed heap size results in *false positive crashes* due to heap memory exhaustion), (ii) preventing users from selecting an incorrect metadata structure, (iii) using only one metadata structure (e.g., min-shadow memory) may miss the opportunity to further improve throughput, as, in some cases, RB tree (or some future metadata structure) may be faster than min-shadow memory; (iv) manually selecting a metadata structure requires extra effort (e.g., measuring each metadata structure's efficiency for the target application), which dynamic metadata structure switching mode avoids by automatically selecting the optimal metadata structure.

Table 4.7.: Evaluating FuZZan's total execution number and unique discovered path for 24 hours fuzz testing with provided seeds. The (M) denotes 1,000,000 (one million) and ratio (%) is the ratio between ASan and FuZZan.

Programs	**Native**		**ASan**		**FuZZan**	
	exec #	**path** #	**exec** #	**path** #	**exec** # (%)	**path** # (%)
cxxfilt	86M	2,769	33M	2,442	51M (55%)	2,651 (9%)
file	29M	1,126	7M	763	9M (29%)	845 (11%)
nm	51M	1,272	7M	822	12M (71%)	872 (6%)
objdump	95M	883	15M	567	17M (13%)	595 (5%)
pngfix	36M	971	18M	912	33M (83%)	982 (8%)
size	52M	703	17M	626	32M (88%)	656 (5%)
tcpdump	70M	3,587	11M	1,540	20M (82%)	2,032 (32%)
Total	419M	11,311	108M	7,672	174M (61%)	8,633 (13%)

4.4.4 Real-world fuzz testing

Our experiments validating FuZZan use a record/replay approach to avoid any impact of randomness, allowing meaningful comparisons to a baseline. However, real-world fuzzing is highly stochastic, and so we also evaluate FuZZan in the context of several real-world end-to-end fuzzing campaigns without deterministic input record/replay. For this experiment, we select the following widely used programs: `cxxfilt`, `nm`, `objdump`, `size` (all from binutil-2.31), `file` (version 5.35), `pngfix` (from libpng 1.6.38) and `tcpdump` (version 4.10.0). Klees et al. [110] select and test `cxxfilt`, `nm`, and `objdump` in their fuzzing evaluation study. The remaining four programs (`size`, `file`, `pngfix`, and `tcpdump`) are widely tested by recent fuzzing works [112–117]. For each binary, we run a fuzzing campaign. Each campaign is conducted for 24 hours and repeated five times. We measure the number of total executions and discovered unique paths when fuzzing with seeds from the seed corpus of each program with the right type file and three different configurations: native, ASan, and FuZZan's dynamic metadata structure switching mode, and report the mean over the five campaigns.

As a result, FuZZan improves throughput over ASan by 61% (up to 88%). Interestingly, FuZZan discovers 13% more unique paths given the same 24 hours time due to improved throughput. Our evaluation also shows that improved throughput increases the possibility of finding more bugs in the same amount of time, as we discuss next.

4.4.5 Bug finding effectiveness

FuZZan increases throughput while maintaining ASan's bug detection capability, potentially enabling it to find more bugs. To demonstrate this, we evaluate FuZZan's bug finding speed and compare it to a fuzzing campaign with ASan. In this evaluation, we target five applications in Google's fuzzer test suite. These applications are chosen because we found bugs in them (using ASan and dynamic metadata structure

Table 4.8.: Evaluating FuZZan's bug finding speed. The TTE denotes the mean time-to-exposure. The AF is assertion error and the BO denotes buffer overflow.

Programs	**ASan**	**FuZZan**		**Type (source)**
	TTE (s)	**TTE (s)**	**rate (%)**	
c-ares	45	25	46	BO (ares_create_query.c:196)
json	29	11	61	AF (fuzzer-parse_json.cpp:50)
libxml2	7,314	4,194	43	BO (CVE-2015-8317)
openssl-1.0.1f	443	336	24	BO (t1_lib.c:2586)
pcre2	7,056	4,020	43	BO (pcre2_match.c:5968)
Total	14,887	8,586	42	-

Table 4.9.: Comparison between Native, MSan, MSan-nolock, and min-shadow memory execution overhead during input record and replay fuzz testing with provided seed sets. MSan-nolock disables lock/unlock for MSan's logging depots. Time (s) indicates the average of execution time. Positive percentages denote overhead, negative percentages denote speedup.

Modes	**time (s)**	**vs. Native (%)**	**vs. MSan (%)**	**vs. MSan nolock (%)**
Native	146	-	-	-
MSan	14,074	9,575	-	-
MSan-nolock	386	165	-97	-
Min-16G	335	130	-98	-13

switching mode) within a 24 hour fuzzing campaign. We use the seeds provided by the test suite and repeated each campaign five times. Note that we do not replay recorded inputs during these campaigns, instead letting the fuzzer generate random inputs. Table 4.8 shows the mean time (over five campaigns) to find each bug. Notably, FuZZan finds all bugs up to 61% (mean 42%) faster than ASan, and is faster in all cases. This experiment emphasizes our belief that throughput is paramount when fuzzing with sanitizers.

4.4.6 FuZZan Flexibility

Appling FuZZan to Memory Sanitizer. Like ASan, numerous sanitizers use shadow memory for their metadata structure [83]. For example, other popular sanitizers, such as Memory Sanitizer (MSan) [99] and Thread Sanitizer (TSan) [100], also rely on shadow memory for metadata. FuZZan optimizes sanitizer usage of shadow memory *without* modifying the stored shadow information or how the sanitizer uses that information. Consequently, porting our shadow metadata improvements in FuZZan from ASan to other sanitizers is a simple engineering exercise. To demonstrate this,

we port FuZZan to MSan. In so doing, we shrink MSan's memory space to implement min-shadow memory 16G for MSan (1GB for the stack, 16GB for the heap, and 2GB for the BSS, data, and text sections combined). We only implement one metadata mode for our MSan proof-of-concept to validate our claim that applies FuZZan to other shadow memory based sanitizers is an engineering exercise.

Table 4.9 summarizes MSan's performance overhead on different modes for all 26 evaluated applications. Initially, min-shadow memory shows high overhead—around 96 times native. Analyzing this, we found that MSan's `fork()` interceptor locks all logging depots before `fork()` and similarly unlocks them afterwards to avoid deadlocks. However, as explained in section 4.3, locking/unlocking logging depots is unnecessary for fuzzing because these logging depots exist for bug reporting and fuzzing inherently enables replay by storing test inputs when the fuzzer finds bugs. We thus disable these lock/unlock functions to create the MSan-nolock mode, which has reasonable overhead (2.6 times that of native).

FuZZan's MSan min-shadow memory 16G mode shows 13% performance improvement compared to MSan-nolock mode, demonstrating FuZZan's efficacy when applied to MSan. We expect that additional optimization and the application of the dynamic switch mode will lead to even higher performance improvement. We leave this engineering as future work.

Applying FuZZan to MOpt-AFL. FuZZan is not coupled to a particular fuzzer or fuzzer version. Most modern fuzzers [109,109,112,118] extend AFL, so our approach applies broadly. To demonstrate this, we apply FuZZan to MOpt-AFL [109], which is an efficient mutation scheduling scheme to achieve better fuzzing efficiency. We modify MOpt-AFL to add FuZZan's profiling feedback and dynamic metadata switching functions. To measure FuZZan's impact on MOpt-AFL, we select seven real-world applications (the same set as Table 4.7) and fuzz them for 24 hours each, repeating the experiment five times to control for randomness in the results. On average, ASan-MOpt-AFL mode discovers 85% more unique paths given the same 24 hours time due to MOpt-AFL's effectiveness compared to ASan. Notably, FuZZan-MOpt-AFL mode

discovers 112% more unique paths (27% higher than ASan-MOpt-AFL) due to the improved throughput.

4.5 Discussion

In this section, we summarize some potential areas for future work, a possible security extension enabled by FuZZan, and lessons learned in designing FuZZan.

Removing conflicts between sanitizers. ASan's shadow memory scheme conflicts with other sanitizers that are also based on shadow memory, e.g., MSan and TSan. Each sanitizer interprets the shadow memory in a mutually exclusive manner, prohibiting the use of multiple concurrent sanitizers. For example, ASan uses shadow memory as a metadata store, while MSan prohibits access to the same memory range. FuZZan's new metadata structures can be adapted to avoid this conflict, and enable true composition of sanitizers, since we use lightweight, independent metadata structures. Each sanitizer can map its own instance of our metadata structure, and all sanitizers may coexist in a single process. However, some engineering effort is required to port sanitizers to our new metadata structures. An alternate approach would be to have one metadata structure that stores information for all sanitizers. Whether having a unified metadata structure or a metadata structure per sanitizer is more efficient is an interesting research question.

Possible security extension. Unfortunately, ASan's virtual memory requirements directly conflict with fuzzers' abilities to detect certain out-of-memory (OOM) bugs. For example, fuzzers typically limit memory usage to detect OOM errors when parsing malformed input. However, ASan's large virtual memory requirement masks OOM bugs, leaving them undetected because of the difficulty of setting precise memory limits. Consequently, using a compact metadata structure with ASan not only improves performance, but also can enable an extension of ASan's policy to cover OOM bugs.

Lessons Learned. Our initial metadata design leveraged a two-layered shadow memory metadata structure that split metadata lookups into two parts: a lookup into

a top-level metadata structure, followed by a lookup into a second-level metadata structure a la page tables. While this design vastly reduced memory consumption and management overhead, the additional runtime cost per metadata access of the additional indirection resulted in the two-layer structure being slower than ASan in all cases.

For dynamic metadata structure switching, we evaluated two additional policies: (i) utilizing more detailed metadata access information such as each object type's (e.g, stack) metadata access (e.g., insert) count and each operation's microbenchmark results, and (ii) running each metadata mode, measuring their execution time, and selecting the fastest metadata mode. In our evaluation, the additional sampling complexity of these policies outweighed any gains from more precisely selecting a metadata structure.

4.6 Related Work

4.6.1 Reducing Fuzzing Overhead

Several approaches reduce the overhead of fuzzing. One approach is to reduce the execution time of each iteration. AFL supports a deferred fork server which requires a manual call to the fork server. The analyst is encouraged to use the deferred fork server, and manually initiate the fork server as late as possible to reduce, not only overhead from linking and libc initializations, but also overhead from the initialization of the target program. Deferred mode, however, cannot reduce the teardown overhead of heavy metadata structures. AFL's persistent mode and libFuzzer eliminate the overhead from creating a new process. However, these approaches require manual effort, and users must know the target programs. Xu et al. [85] implement several new OS primitives to improve the efficiency of fuzzing on multicore platforms. Especially, by supporting a new system call, `snapshot` instead of `fork`, they reduce the overhead of creating a process. Moreover, they reduce the overhead from file system

contention through a dual file system service. However, this approach requires kernel modifications for the new primitives, and does not reduce the overhead of sanitizers.

Another approach is to improve fuzzing itself so that it can find more crashes within the same amount of executions. AFLFast [112] adopts a Markov chain model to select a seed. If inputs mutated from a seed explore more new paths, the seed has higher probability to be selected. With given target source locations, AFLGo [118] selects a seed that has higher probabilities to reach the source locations. Several approaches adopt hybrid fuzzing, taint analysis, and machine learning to help fuzzers explore more paths. SAVIOR [119] uses hybrid fuzzing, combining it with concolic execution to explore code blocks guarded by complex branch conditions. RedQueen [114] uses taint analysis and symbolic execution for the same purpose. VUzzer [120] also uses dynamic taint analysis and mutates bytes which are related to target branch conditions to efficiently explore paths. TIFF [121] infers the type of the input bytes through dynamic taint analysis and uses the type information to mutate the input. Matryoshka [122] uses both data flow and control flow information to explore nested branches. In addition to hybrid fuzzing with traditional techniques such as symbolic and concolic executions, NEUZZ [117] adapts neural network and sets the number of covered paths as an objective function to maximize covered paths. Angora [115] adapts both taint analysis and a gradient descent algorithm to improve the number of covered paths. These approaches do not reduce the execution time of each iteration. They are therefore orthogonal to our work. Thus, we can use these approaches to further increase fuzzing performance.

4.6.2 Optimizing Sanitizers

Since C/C++ programming languages are memory and type unsafe languages, several sanitizers [83] target memory safety violations [14, 74, 99, 101, 123] and type safety violations [3–5, 35]. Despite their broad use, sanitizers have several limitations

such as high overhead, limited detection abilities, and incompatibility with other sanitizers.

To reduce sanitizer overhead, ASAP [87] and PartiSan [88] disable check instrumentation on the hot path according to their policies. The intuition of both approaches is that most of the sanitizer's overhead comes from checks on a few hot code paths that are frequently executed (e.g., instrumentation in a loop). ASAP removes check instrumentation on the hot path based on pre-calculated profiling results at compile time. In PartiSan [88], Lettner et al., propose runtime partitioning to more effectively remove check instrumentation based on runtime information during execution. However, both approaches miss a main source of overhead when reducing the cost of ASan during fuzzing campaigns: the overhead is due to memory management and not due to the low overhead safety checks. As ASAP and PartiSan target the cost of checks, they are complementary to FuZZan. To fuzz quickly, there is an option to generate a corpus from a normal binary, and then feed the corpus to an ASan binary. FuZZan can also adopt this option for fast fuzzing.

Pina et al., [89] use multi-version execution to concurrently run sanitizer-protected processes together with native processes, synchronizing all versions at the system-call level. To synchronize all versions, they use a system-call buffer and a Domain-Specific Language [90] to resolve conflicts between different program versions. Xu et al., [91] propose Bunshin to reduce the overhead of sanitizers and conflicts based on the N-version system through their check distribution, sanitizer distribution, and cost distribution policies. Since these approaches are based on N-version systems, they increase hardware requirements such as several dedicated cores and at least N times of memory. Also, these approaches do not address the fundamental problem of ASan memory overhead.

4.7 Conclusion

Combining a fuzzer with sanitizers is a popular and effective approach to maximize bug finding efficacy. However, several design choices of current sanitizers hinder fuzzing effectiveness, increasing the runtime cost and reducing the benefit of combining fuzzing and sanitization.

We show that the root cause of this overhead is the heavy metadata structure used by sanitizers, and propose FuZZan to optimize sanitizer metadata structures for fuzzing. We implement and apply these ideas to ASan. We design new metadata structures to replace ASan's rigid shadow memory, reducing the memory management overhead while maintaining the same error detection capabilities. Our dynamic metadata structure adaptively selects the most efficient metadata structure for the current fuzzing campaign without manual configuration.

Our evaluation shows that FuZZan improves performance over ASan 52% when starting with empty seeds (48% with Google's seed corpus). Based on improved throughput, FuZZan discovers 13% more unique paths given the same 24 hours and finds bugs 42% faster. The open-source version of FuZZan is available at `https://github.com/HexHive/FuZZan`.

5 THE DIRECTION OF FUTURE RESEARCH

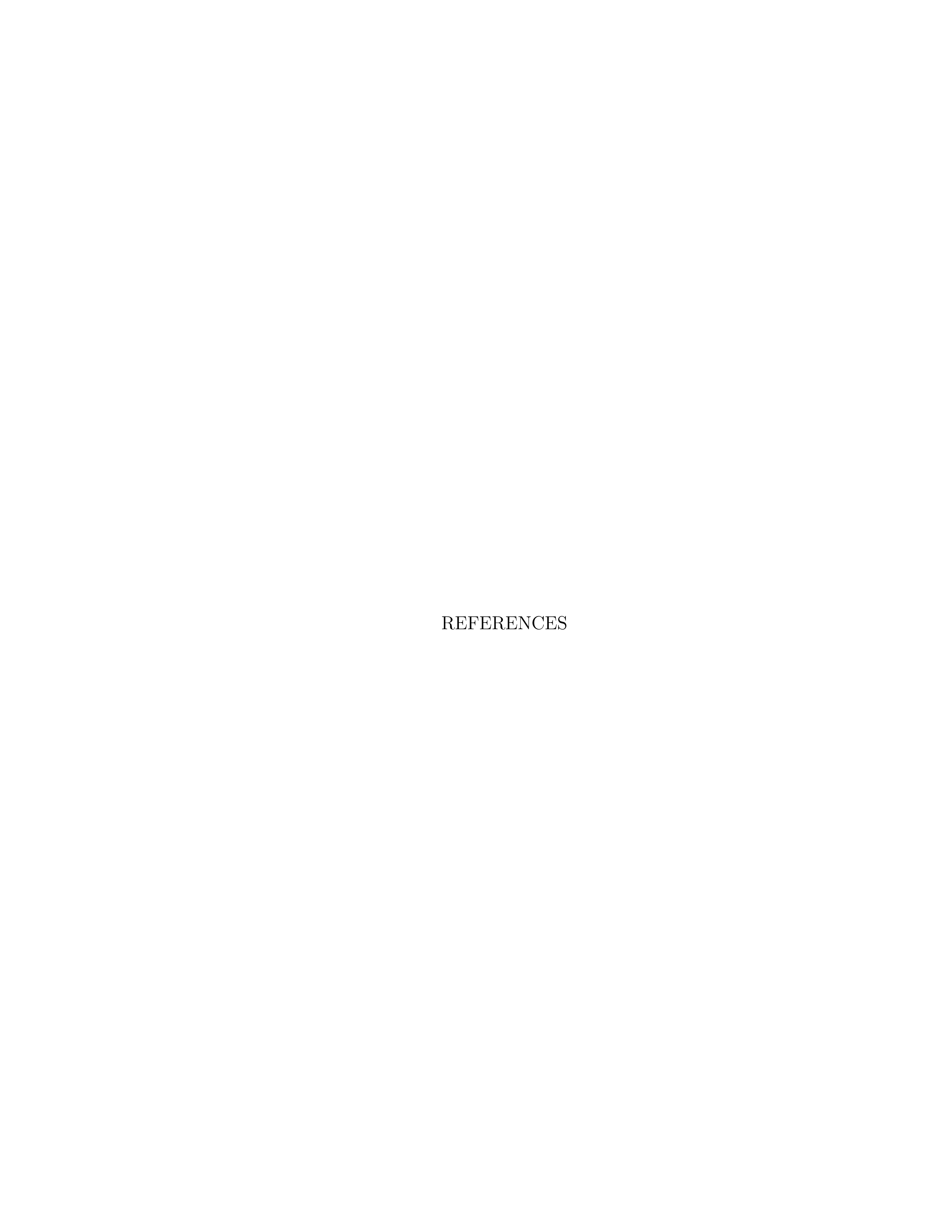
For future research, we plan to focus on type safety research to further improve existing C++ type confusion detectors and detect type confusion problems in cross language boundaries (e.g., foreign function interface).

In terms of improving existing C++ type confusion detectors, we plan to further improve and evaluate the V-Type project. For this, we will evaluate V-Type with more complicated real-world applications such as Firefox and Chromium. We will also address additional compatibility issues, which can occur during the testing of these real-world applications. To find type confusion bugs, we plan to apply V-Type to fuzzing to find bugs effectively.

Additionally, we plan to propose IPCSan, which can detect type confusion problems in cross language boundaries between Rust (type safety language) and C/C++ (type unsafety language). Type safe languages have been developed to replace the type unsafe languages C/C++. These days, one of the popular languages that guarantee type safety is Rust. However, these guarantees are valid only for Rust applications, that is, they may not hold when cross language compiled processes communicate. In this work, we explore how type confusion vulnerabilities can arise in cross language boundaries between Rust and C++. Thus, we will propose IPCSan, a tool for detecting type confusion in cross language boundaries to support cross language type safety. More specifically, IPCSan inserts instrumentation for object tracking to predict expected type and inserts runtime checks to verify whether the received type is equivalent to the type expected.

6 SUMMARY

Computer systems and applications are mainly implemented in C/C++. However, C/C++ trade type and memory safety for performance. Our system, HexType, introduces a novel type confusion detection mechanism, which provides low-overhead disjoint metadata structures, compiler optimizations, and handling specific object allocation patterns. Consequently, compared to prior work, the HexType prototype results show significantly improved detection coverage and reduced performance overhead. Although HexType significantly reduces performance overhead and detection coverage, HexType still has considerable overhead from managing disjoint metadata structure and object tracking and has a false positive rate issue from incorrect object tracking. In regards to our other type confusion detection mechanism, V-Type addresses these issues via forcibly changing non-polymorphic objects into polymorphic ones to make sure all objects maintain type information. Through this method, V-Type eliminates the burden of tracking objects and managing disjoint metadata structure. We have another project, FuZZan, that optimizes sanitizer metadata structures for fuzzing. Because of this, FuZZan improves fuzzing throughput, and this helps the tester to discover more bugs given the same amount of time. Combined, these defenses, HexType, V-Type, and FuZZan, provide the capability for type and memory safety violations to be minimized, substantially increasing the security of C/C++ software.



REFERENCES

REFERENCES

[1] P. Muntean, S. Wuerl, J. Grossklags, and C. Eckert, "Castsan: Efficient detection of polymorphic c++ object type confusions with llvm," in *European Symposium on Research in Computer Security*. Springer, 2018, pp. 3–25.

[2] C. Pang, Y. Du, B. Mao, and S. Guo, "Mapping to bits: Efficiently detecting type confusion errors," in *Proceedings of the 34th Annual Computer Security Applications Conference*. ACM, 2018, pp. 518–528.

[3] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. 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.

[4] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector." in *USENIX Security Symposium*, 2015, pp. 81–96.

[5] L. team, "Tysan: A type sanitizer," 2018, [Online; accessed 11-May-2019]. [Online]. Available: https://reviews.llvm.org/D32199

[6] G. J. Duck and R. H. Yap, "Effectivesan: type and memory error detection using dynamically typed c/c++," in *ACM SIGPLAN Notices*, vol. 53, no. 4. ACM, 2018, pp. 181–195.

[7] G. Dmitry Vyukov, "Address/thread/memorysanitizer slaughtering c++ bugs," https://www.slideshare.net/sermp/sanitizer-cppcon-russia, Online; accessed 11-May-2019.

[8] A. Konovalov, "Kerneladdresssanitizer (kasan) a fast memory error detector for the linux kernel," https://events.static.linuxfound.org/sites/events/files/slides/LinuxConNorthAmerica2015KernelAddressSanitizer.pdf, Online; accessed 11-May-2019.

[9] Google, "Addresssanitizerfoundbugs," https://github.com/google/sanitizers/wiki/AddressSanitizerFoundBugs, Online; accessed 11-May-2019.

[10] M. Miller, "Trends, challenge, and shifts in software vulnerability mitigation," https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf, Accessed 15-May-2019.

[11] R. Hastings and B. Joyce, "Purify: Fast detection of memory leaks and access errors," in *In Proc. of the Winter 1992 USENIX Conference*. Citeseer, 1991.

[12] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision." in *USENIX Annual Technical Conference, General Track*, 2005, pp. 17–30.

[13] D. Bruening and Q. Zhao, "Practical memory checking with dr. memory," in *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2011, pp. 213–223.

[14] G. Project, "Address sanitizer," https://github.com/google/sanitizers/wiki/AddressSanitizer, Online; accessed 11-May-2019.

[15] B. Perens, "Electric fence malloc debugger," *Pixar Animation Studios*, 1993.

[16] R. W. Jones and P. H. Kelly, "Backwards-compatible bounds checking for arrays and pointers in c programs," in *Proceedings of the 3rd International Workshop on Automatic Debugging; 1997 (AADEBUG-97)*, no. 001. Linköping University Electronic Press, 1997, pp. 13–26.

[17] D. Dhurjati and V. Adve, "Efficiently detecting all dangling pointer uses in production servers," in *International Conference on Dependable Systems and Networks (DSN'06)*. IEEE, 2006, pp. 269–280.

[18] T. H. Dang, P. Maniatis, and D. Wagner, "Oscar: A practical page-permissions-based scheme for thwarting dangling pointers," in *26th {USENIX} Security Symposium ({USENIX} Security 17)*, 2017, pp. 815–832.

[19] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector." in *NDSS*, vol. 2004, 2004, pp. 159–169.

[20] D. Dhurjati and V. Adve, "Backwards-compatible array bounds checking for c with very low overhead," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 162–171.

[21] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors." in *USENIX Security Symposium*, 2009, pp. 51–66.

[22] F. C. Eigler, "Mudflap: Pointer use checking for c/c+," *Proceedings of the First Annual GCC Developers' Summit*, pp. 57–70, 2003.

[23] Y. Younan, P. Philippaerts, L. Cavallaro, R. Sekar, F. Piessens, and W. Joosen, "Paricheck: an efficient pointer arithmetic checker for c programs," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*. ACM, 2010, pp. 145–156.

[24] G. J. Duck and R. H. Yap, "Heap bounds protection with low fat pointers," in *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 2016, pp. 132–142.

[25] G. J. Duck, R. H. Yap, and L. Cavallaro, "Stack bounds protection with low fat pointers." in *NDSS*, 2017.

[26] S. C. Kendall, "Bcc: Runtime checking for c programs," in *Proceedings of the USENIX Summer Conference*, 1983, pp. 5–16.

[27] J. L. Steffen, "Adding run-time checking to the portable c compiler," *Software: Practice and Experience*, vol. 22, no. 4, pp. 305–316, 1992.

[28] T. M. Austin, S. E. Breach, and G. S. Sohi, *Efficient detection of all pointer and array access errors*. ACM, 1994, vol. 29, no. 6.

[29] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in c programs," *Software: Practice and Experience*, vol. 27, no. 1, pp. 87–110, 1997.

[30] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 245–258, 2009.

[31] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy code," in *ACM SIGPLAN Notices*, vol. 37, no. 1. ACM, 2002, pp. 128–139.

[32] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c." in *USENIX Annual Technical Conference, General Track*, 2002, pp. 275–288.

[33] W. Xu, D. C. DuVarney, and R. Sekar, "An efficient and backwards-compatible transformation to ensure memory safety of c programs," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 6, pp. 117–126, 2004.

[34] N. Nethercote and J. Fitzhardinge, "Bounds-checking entire programs without recompiling," *SPACE*, 2004.

[35] Y. Jeon, P. Biswas, S. Carr, B. Lee, and M. Payer, "Hextype: Efficient detection of type confusion errors for c++," in *CCS*, 2017.

[36] Y. Jeon, J. Rhee, C. H. Kim, Z. Li, M. Payer, B. Lee, and Z. Wu, "Polper: Process-aware restriction of over-privileged setuid calls in legacy applications," in *Proceeding of ACM Conf on Data and Application Security and Privacy (CODASPY)*, 2019.

[37] Clang, "Clang 3.9 documentation - control flow integrity," http://clang.llvm.org/docs/ControlFlowIntegrity.html, Online; accessed 17-May-2017.

[38] G. Project, "Undefinedbehavior sanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, Online; accessed 11-May-2019.

[39] I. Haller, E. Goktas, E. Athanasopoulos, G. Portokalidis, and H. Bos, "Shrinkwrap: Vtable protection without loose ends," in *ACSAC*, 2015.

[40] C. Zhang, S. A. Carr, T. Li, Y. Ding, C. Song, M. Payer, and D. Song, "Vtrust: Regaining trust on virtual calls," in *NDSS*, 2016.

[41] B. Lee, C. Song, T. Kim, and W. Lee, "Type casting verification: Stopping an emerging attack vector," in *USENIX Security*, 2015. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/lee

[42] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos, and E. van der Kouwe, "Typesan: Practical type confusion detection," in *23rd ACM SIGSAC Conference on Computer and Communications Security*, 2016.

[43] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in gcc & llvm," in *USENIX Security*, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2671225.2671285

[44] V. van der Veen, E. Goktas, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, "A tough call: Mitigating advanced code-reuse attacks at the binary level," in *IEEE S&P*, 2016.

[45] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, "Practical Context-Sensitive CFI," in *CCS*, 2015.

[46] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *OSDI*, 2014.

[47] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *ACM Sigplan Notices*, vol. 44, no. 6. ACM, 2009, pp. 245–258.

[48] ——, "Cets: compiler enforced temporal safety for c," in *ACM Sigplan Notices*, vol. 45, no. 8. ACM, 2010, pp. 31–40.

[49] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX, 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[50] Microsoft, "Microsoft security intelligence report," https://www.microsoft.com/security/sir, Online; accessed 17-May-2017.

[51] CERT, "the cert c++ coding standard (5 the void section)," https://www.securecoding.cert.org/confluence/display/cplusplus/5+The+Void/, Online; accessed 17-May-2017.

[52] JTC1/SC22/WG21, "Iso/iec 14882:2014 programming language c++," http://www.iso.org/iso/catalogue_detail.htm?csnumber=64029, Online; accessed 17-May-2017.

[53] llvm, "The llvm compiler infrastructure project," http://llvm.org/, Online; accessed 11-May-2019.

[54] S. P. E. Corporation, "Spec cpu 2006," http://www.spec.org/cpu2006, Online; accessed 17-May-2017.

[55] T. M. Foundation, "Mozilla firefox," https://www.mozilla.org/firefox, Online; accessed 17-May-2017.

[56] Google, "Octane benchmark," https://code.google.com/p/octane-benchmark, Online; accessed 17-May-2017.

[57] T. M. Foundation, "Dromaeo, javascript performance testing," https://www.webkit.org/perf/sunspider/sunspider.html, Online; accessed 17-May-2017.

[58] Q. C. Review, "Type confusion: From qmapnodebase to qmapnode," https://codereview.qt-project.org/#/c/191188/, Online; accessed 17-May-2017.

[59] M. Zalewski, "American fuzzy lop." http://lcamtuf.coredump.cx/afl, Online; accessed 11-May-2019.

[60] "Webkit css type confusion," http://em386.blogspot.com/2010/12/webkit-css-type-confusion.html, Online; accessed 17-May-2017.

[61] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *CCS*, 2005.

[62] J. Criswell, N. Dautenhahn, and V. Adve, "KCoFI: Complete control-flow integrity for commodity operating system kernels," in *Oakland: IEEE Symp. on Security and Privacy*, 2014.

[63] X. Ge, N. Talele, M. Payer, and T. Jaeger, "Fine-Grained Control-Flow Integrity for Kernel Software," in *EuroSP: IEEE European Symp. on Security and Privacy*, 2016.

[64] M. Zhang and R. Sekar, "Control flow integrity for cots binaries," in *SEC: USENIX Security Symposium*, 2013.

[65] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Computing Surveys*, vol. 50, no. 1, 2018, preprint: https://arxiv.org/abs/1602.04056.

[66] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos, "Vtpin: practical vtable hijacking protection for binaries," in *Proceedings of the 32nd Annual Conference on Computer Security Applications.* ACM, 2016, pp. 448–459.

[67] D. Jang, Z. Tatlock, and S. Lerner, "Safedispatch: Securing c++ virtual calls from memory corruption attacks." in *NDSS*, 2014.

[68] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song, "Vtint: Protecting virtual function tables' integrity." in *NDSS*, 2015.

[69] S. A. Carr and M. Payer, "DataShield: Configurable Data Confidentiality and Integrity," in *AsiaCCS: ACM Symp. on InformAtion, Computer and Communications Security*, 2017.

[70] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker." in *USENIX Annual Technical Conference*, 2012, pp. 309–318.

[71] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *ACM Sigplan notices*, vol. 42, no. 6. ACM, 2007, pp. 89–100.

[72] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy software," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 27, no. 3, pp. 477–526, 2005.

[73] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 351–366.

[74] B. Lee, C. Song, Y. Jang, T. Wang, T. Kim, L. Lu, and W. Lee, "Preventing use-after-free with dangling pointers nullification." in *NDSS*, 2015.

[75] A. Loginov, S. H. Yong, S. Horwitz, and T. Reps, "Debugging via run-time type checking," in *International Conference on Fundamental Approaches to Software Engineering*. Springer, 2001, pp. 217–232.

[76] L. Developers, "Tysan: A type sanitizer," https://lists.llvm.org/pipermail/llvm-dev/2017-April/111766.html.

[77] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[78] B. Stroustrup, "How do i define an in-class constant," https://www.stroustrup.com/bs_faq2.html#in-class.

[79] L. Developers, "libc++, c++ standard library," https://libcxx.llvm.org/.

[80] G. Developers, "The gnu c++ library," https://gcc.gnu.org/onlinedocs/libstdc++/.

[81] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, 1990.

[82] Google, "Clusterfuzz," https://google.github.io/clusterfuzz/, [Online; accessed 11-May-2019].

[83] D. Song, J. Lettner, P. Rajasekaran, Y. Na, S. Volckaert, P. Larsen, and M. Franz, "Sok: Sanitizing for security," *arXiv preprint arXiv:1806.04355*, 2018.

[84] T. C. Project, "Addresssanitizer (asan)," https://www.chromium.org/developers/testing/addresssanitizer, Online; accessed 11-May-2019.

[85] W. Xu, S. Kashyap, C. Min, and T. Kim, "Designing new operating primitives to improve fuzzing performance," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: ACM, 2017, pp. 2313–2328. [Online]. Available: http://doi.acm.org/10.1145/3133956.3134046

[86] L. team, "libfuzzer – a library for coverage-guided fuzz testing," 2018, [Online; accessed 11-May-2019]. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[87] J. Wagner, V. Kuznetsov, G. Candea, and J. Kinder, "High system-code security with low overhead," in *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE, 2015, pp. 866–879.

[88] J. Lettner, D. Song, T. Park, S. Volckaert, P. Larsen, and M. Franz, "Partisan: Fast and flexible sanitization via run-time partitioning," *arXiv preprint arXiv:1711.08108*, 2017.

[89] L. Pina, A. Andronidis, and C. Cadar, "Freeda: Deploying incompatible stock dynamic analyses in production via multi-version execution," *System*, vol. 9, no. 10, p. 11, 2018.

[90] L. Pina, D. Grumberg, A. Andronidis, and C. Cadar, "A dsl approach to reconcile equivalent divergent program executions," in *USENIX ATC*, vol. 17, 2017.

[91] M. Xu, K. Lu, T. Kim, and W. Lee, "Bunshin: Compositing security mechanisms through diversification," in *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, 2017, pp. 271–283.

[92] K. Serebryany, "Sanitize, fuzz, and harden your c++ code," https://www.usenix.org/sites/default/files/conference/protected-files/enigma_slides_serebryany.pdf, Online; accessed 11-May-2019.

[93] ——, "Sanitize, fuzz, and harden your c++ code," https://github.com/google/sanitizers/blob/master/hwaddress-sanitizer/HardwareMemoryTaggingtomakeC_C++memorysafe(r)-iSecCon2018.pdf, Online; accessed 11-May-2019.

[94] D. Vyukov, "Syzbot," https://syzkaller.appspot.com/upstream.

[95] google team, "fuzzer test suite," 2018, [Online; accessed 11-May-2019]. [Online]. Available: https://github.com/google/fuzzer-test-suite

[96] M. Zalewski, "New in AFL: persistent mode," https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html.

[97] Google, "Libfuzzer tutorial," https://github.com/google/fuzzer-test-suite/blob/master/tutorial/libFuzzerTutorial.md.

[98] V. J. M. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, 2019.

[99] E. Stepanov and K. Serebryany, "Memorysanitizer: fast detector of uninitialized memory use in c++," in *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2015, pp. 46–55.

[100] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: data race detection in practice," in *Proceedings of the workshop on binary instrumentation and applications*. ACM, 2009, pp. 62–71.

[101] E. Van Der Kouwe, V. Nigade, and C. Giuffrida, "Dangsan: Scalable use-after-free detection," in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 405–419.

[102] R. Hastings, "Purify: Fast detection of memory leaks and access errors," in *Proceedings of the USENIX Security Symposium (SEC)*, 1992.

[103] N. Hasabnis, A. Misra, and R. Sekar, "Light-weight bounds checking," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2012.

[104] Y. Younan, "FreeSentry: protecting against use-after-free vulnerabilities due to dangling pointers," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[105] L. kernel document, "The kernel address sanitizer (kasan)," https://www.kernel.org/doc/html/v4.14/dev-tools/kasan.html, [Online; accessed 11-May-2019].

[106] Wikipedia, "x32 ABI," https://en.wikipedia.org/wiki/X32_ABI.

[107] V. Jacobson, "Congestion avoidance and control," *ACM SIGCOMM computer communication review*, 1988.

[108] T. Kroes, K. Koning, C. Giuffrida, H. Bos, and E. van der Kouwe, "Fast and generic metadata management with mid-fat pointers," in *Proceedings of the European Workshop on Systems Security (EuroSec)*, 2017.

[109] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized Mutation Scheduling for Fuzzers," in *Proceedings of the USENIX Security Symposium (SEC)*, 2019.

[110] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 2123–2138.

[111] NIST, "Juliet test suite," 2017, [Online; accessed 11-May-2019]. [Online]. Available: https://samate.nist.gov/SARD/testsuite.php

[112] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1032–1043.

[113] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 627–637.

[114] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "REDQUEEN: Fuzzing with Input-to-State Correspondence," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[115] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," *arXiv preprint arXiv:1803.01307*, 2018.

[116] H. Peng, Y. Shoshitaishvili, and M. Payer, "T-Fuzz: fuzzing by program transformation," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2018.

[117] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2019.

[118] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 24th ACM Conference on Computer and Communications Security, ser. CCS*, 2017, pp. 1–16.

[119] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, L. Lu *et al.*, "SAVIOR: Towards Bug-Driven Hybrid Testing," in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2020.

[120] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[121] V. Jain, S. Rawat, C. Giuffrida, and H. Bos, "TIFF: Using Input Type Inference To Improve Fuzzing," in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2018.

[122] P. Chen, J. Liu, and H. Chen, "Matryoshka: Fuzzing Deeply Nested Branches," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2019.

[123] N. Burow, D. McKee, S. A. Carr, and M. Payer, "Cup: Comprehensive user-space protection for c/c++," in *Proceedings of the 2018 on Asia Conference on Computer and Communications Security.* ACM, 2018, pp. 381–392.

VITA

VITA

Yuseok Jeon received his Bachelor of Science (BS) in Computer Science from INHA University, South Korea, in August 2007, and the Master of Science (MS) in Computer and Communications Engineering from POSTECH, South Korea, in February 2010. He has worked for the National Security Research Institute, Samsung, NEC Labs America, and Intel. His primary interests are in solving software and systems security problems via programming analysis.