Adapting software testing techniques to enhance software security

Haller, I.

2017

document version
Publisher's PDF, also known as Version of record

Adapting Software Testing Techniques To Enhance Software Security

Ph.D. Thesis

István Haller

Vrije Universiteit Amsterdam, 2017

vrije Universiteit amsterdam

This work was funded by the Microsoft Research Ph.D. Scholarship Programme through the project MRL2011-049.

ISBN 978-94-6295-593-6

Cover design by István Haller.
Printed by ProefschriftMaken.

VRIJE UNIVERSITEIT

ADAPTING SOFTWARE TESTING TECHNIQUES TO ENHANCE SOFTWARE SECURITY

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. V. Subramaniam,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op maandag 3 april 2017 om 11.45 uur
in de aula van de universiteit,
De Boelelaan 1105

door

István Haller

geboren te Tirgu Mures, Roemenië

promotor: prof.dr.ir. H.J. Bos
copromotor: dr. C. Giuffrida

"For millions of years flowers have been producing thorns. For millions of years sheep have been eating them all the same. And it's not serious, trying to understand why flowers go to such trouble to produce thorns that are good for nothing?"

Antoine de Saint-Exupéry, The Little Prince.

Acknowledgements

As I was ready to leave the beautiful city of Amsterdam to pursue a Ph.D. somewhere else, several people recommended that I have a short chat with Herbert Bos about his research. I agreed to the discussion, mostly to satisfy their curiosity, because I did not think that I could not contribute anything valuable to his work. But as soon as I left his office, I knew that my time in Amsterdam would be extended by many more years. Thank you Herbert for taking a risk with me and for being such a great leader to me during these years. At the same time, the experience would not have been the same without the help of Asia Slowinska. Her perseverance and drive for perfection shaped my development as a researcher and I will continue to strive for the ideals that she passed onto me. Thank you Asia for being a role model and a friend at the same time. Leadership-wise I cannot forget about Cristiano Giuffrida and Erik van der Kouwe. You helped me revitalize my research when it hit rock bottom, through your awesome ideas and guidance. I'm just sorry that we did not have the chance to collaborate more, but I am hopeful that the future holds some surprises in this regard.

Next, I would like to express my gratitude to all the members of my Ph.D. thesis committee: Andy Tanenbaum, Thorsten Holz, Frank Piessens, Cristian Cadar and Manuel Costa. Their valuable comments greatly contributed to improving the quality of this dissertation. I am especially grateful to Andy Tanenbaum, for encouraging me to pursue this PhD opportunity and for the constant support throughout the experience. I am also extremely grateful to Thorsten Holz, Frank Piessens and Cristian Cadar for all the opportunities I had in gaining personal insights about their research, which allowed me to enhance mine, and to Manuel Costa for helping me transfer all the skills I gained over these years into practical experience.

The experience of the Ph.D. would not have been the same without the Systems and Network Security group offering many opportunities for fun and creative times

together. It was a privilege to work and befriend so many smart people in such a small space. Thank you Lucian, Chen, Enes, Erik, Remco, Angelos, Andrei, Tadeüs, Koen, Sanjay, Radhesh, Dennis, Koustubha, Manolis, Kaveh and Ben.

I would also like to thank all the current and former members of the Computer Systems group, who were always available to brainstorm or just to share cool stories. A special thanks goes out to Caroline, who was always there for me, no matter how much I messed up with paperwork, making her work difficult.

Amsterdam will always keep a special place in my heart, mostly thanks to the Romanian student community, which took me in from day one and allowed for the most memorable 6 years of my life. I will never forget the parties until 7AM or the Easter feasts that we organized together to keep ourselves sane and smiling even when it was hard. It was amazing to see so many people who could come together and support each other. While I don't want to single out people too much, I do have to mention a couple of very special people from this group: Cristina, Andreea, Gentiana, Florina and Calin.

Besides the experience in Amsterdam, I have to acknowledge all the guidance I received towards research back in Romania. A big thank you goes out to the members of the Image Processing and Pattern Recognition group lead by professor Sergiu Nedevschi, who gave me the first insights into top quality research. At the same time, I would also like to thank professor Rodica Potolea for helping me find the right path in the complicated world of academia.

Of course, none of this would have been possible without my wonderful parents Piroska and Istvan. You have seen the potential in me very early on and helped develop it without ever making it feel like pressure. Thank you for everything in life!

István Haller
Amsterdam, The Netherlands, January 2017

Contents

List of Figures

List of Tables

Publications

This dissertation includes a number of research papers, as appeared in the following conference proceedings [1]:

Istvan Haller, Asia Slowinska, Matthias Neugschwandtner and Herbert Bos. Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations[2]. In *Proceedings of the 22nd USENIX Security Symposium (Usenix Security 13)*, pages 49–64, 2013, Washington DC, USA.

Istvan Haller, Asia Slowinska and Herbert Bos. Scalable data structure detection and classification for C/C++ binaries[3]. In *Empirical Software Engineering Journal*, pages 1–33, 2015, Springer.

Istvan Haller, Enes Göktaş, Elias Athanasopoulos, Georgios Portokalidis and Herbert Bos. ShrinkWrap: VTable Protection Without Loose Ends[4]. In *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC 2015)*, pages 341–350, 2015, Los Angeles, USA.
Awarded Best Student Paper.

Istvan Haller, Erik van der Kouwe, Cristiano Giuffrida and Herbert Bos. METAlloc: Efficient and Comprehensive Metadata Management for Software Security Hardening[5]. In *Proceedings of the 9th European Workshop on System Security (EuroSec 2016)*, pages 5:1–5:6, 2016, London, United Kingdom.

[1] The text differs from the original version in minor editorial changes made to improve readability.
[2] Appears in Chapter 2.
[3] Appears in Chapter 3.
[4] Appears in Chapter 4.
[5] Appears in Chapter 5.

Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida and Herbert Bos, Erik van der Kouwe. TypeSan: Practical Type Confusion Detection[6]. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 517-528, 2016, Vienna, Austria.

Related publications not included in the dissertation are listed in the following:

Istvan Haller, Asia Slowinska, Matthias Neugschwandtner and Herbert Bos. Dowser: a guided fuzzer to find buffer overflow vulnerabilities In *Proceedings of the 6th European Workshop on Systems Security (EuroSec 13)*, 2013, Prague, Czech Republic.

Istvan Haller, Asia Slowinska and Herbert Bos. MemPick: High-level data structure detection in C/C++ binaries In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE 2013)*, pages 32–41, 2013, Koblenz, Germany.

[6] Appears in Chapter 6.

1 Introduction

Software security describes the ability of software to disallow malicious users to trigger undesirable behavior from the perspective of the user or the developer. Such behavior is not part of the software specification and is typically triggered by leveraging bugs in the software design or implementation. Malicious developers can also introduce this behavior unbeknown to their users, however in this dissertation we will focus on the ability of benign software developers to keep their systems running per their desired specifications. The process of using a software bug to trigger unexpected behavior is called exploitation and the bugs involved in this process are called vulnerabilities. Famous recent examples of simple software bugs with major security implication include the OpenSSL Heartbleed bug, the Android StageFright vulnerability and the yearly Pwn2Own contest where hackers showcase the ability to leverage bugs within he major browsers (Chrome, FireFox, Internet Explorer) to hack users. These examples occurred against the best efforts of developers to deploy state-of-the art software testing techniques and significant computational resources [18; 10] to find the bugs before the malicious actors do. This shows that existing software testing techniques are not sufficient alone to ensure the security of complex software and further hardening techniques need to be deployed to keep the bugs from transforming into vulnerabilities. In this dissertation, we explore the potential to minimize the impact of vulnerabilities, starting from targeted testing for key software bugs all the way to pin-point fail-safes for certain software bugs.

Memory corruption bugs represent one of the key vulnerability vectors used malicious actors to this day. For example, the real-world issues presented previously all fall into this category. Exploiting memory corruption bugs has been first presented 20 years ago [101], yet studies [128; 125] show that they are not going away in the short-term. This class of bugs are an artifact of unsafe programming languages, which allow the code to manipulate memory pointers explicitly, without validation. While newer programing languages (such as Rust, Java, C#) are designed to eliminate memory corruption bugs entirely, there is still a large body of software built on

top of unsafe languages, including all the popular browsers and operating systems. For this reason, we focus on memory corruption bugs in this dissertation.

At first sight, comprehensive software testing seems like the best alternative to eliminate all memory corruption bugs in a piece of code. While code analysis can uncover a good portion of software defects, the unstructured nature of unsafe languages makes it impossible to make formal guarantees when dealing with these languages. Even entities heavily invested in software reliability, such as NASA [106] primarily rely on software testing versus more formal methods when searching for defects. But eliminating all memory corruption bugs, would require executing the program with all possible input combinations, a task exponential in the size of the input space. For this reason, the process of software testing typically involves heuristics to maximize effectiveness given a set of computational resources. Effectiveness can be measured either in terms of the coverage achieved across different types of code artifacts or by counting the number of discovered bugs. The former is an indirect measure of the effectiveness, as bugs can still reside in the uncovered artifacts, but it is always applicable and consistent as software evolves. While the number of bugs is a direct measure of the effectiveness for a certain test iteration, it does not offer any baseline value, since the total number of bugs in a piece of code is unknown. The number of discoverable bugs also fluctuates as software evolves and bugs are fixed. Thus, most systems are designed to maximize coverage first and foremost. Unfortunately, we found as part of this dissertation, that such coverage based heuristics are particularly poorly suited when searching for certain types of vulnerabilities. Furthermore, the existing heuristics were designed with users in mind, attempting to maximize the functionalities and user inputs expected during normal execution. In contrast, security deals with malicious actors who focus on uncommon, untested behavior. Many exploitable bugs relate to functionality which is triggered rarely and thus easily avoided during testing [131].

The best way to mitigate the impact of a memory corruption bug, missed during testing, is to restrict the capabilities that an attacker can leverage during exploitation. Control-flow integrity is the most promising research direction [3; 141; 139; 127; 99; 37] in this area, attempting to enforce restrictions on the programs control-flow based on its explicit specification. Memory corruption bugs grant the ability to corrupt the data, but most exploits these days also require changes to the control-flow of the program, steering it towards interesting behavior. Control-flow integrity attempts to extract specifications about the program's control-flow either from the source code or the binary and to enforce it at run-time. The primary issue with control-flow integrity is the lack of standardized approaches to extract this specification. Researchers started out with course-grained approaches which used only a rough over-approximation of the control-flow specification [141; 139], but this was shortly proven to be ineffective against determined attackers [54; 55; 38]. This spawned a cat and mouse game of researchers racing to develop more refined variants of control-flow integrity [127; 99] while also trying to exploit the solutions presented by others [113; 30; 44; 23]. Furthermore, the complexity of the proposed

solutions makes them susceptible to implementation and design bugs of their own. This dissertation as well as parallel work found such problems [63; 30] even within solutions designed and developed by major software vendors. These issues severely affect the confidence in control-flow integrity solutions, undermining its deployment to the general public and making it hard to evaluate its full potential.

With so much research going towards ensuring control-flow integrity, some researchers started to investigate a new generation of exploits which rely only on data corruption [23; 70]. These attacks rely on interesting functionality available along the expected execution path of the program. Attackers manipulate the data space (stack, heap, globals) to change the side-effects of said execution path, triggering unexpected behavior. While well designed control-flow integrity remains a valuable solution against existing attacks, new defenses need to be developed to tackle this new class of attacks. These defenses must tackle the root problem of memory corruption inherent to unsafe languages. Literature already contains several instrumentation techniques [115; 137; 84; 7; 6], which are very effective at detecting individual classes of memory corruption bugs, but they incur too much run-time overhead to be applied at the end user. In practice these techniques become relegated as testing tools to be used internally within development companies, but attackers can still leverage undiscovered bugs when targeting users. Their overhead is composed of two factors, one is the need to track information at run-time (such as object type, object size, etc.) and the other is a series of checks introduced throughout the code to validate the integrity of the data. To minimize the impact of the latter, researchers have proposed probabilistic defenses, which scale the checks with the desired software overhead [131]. This allows a software developer to balance the security guarantees with the performance requirements of the user. Unfortunately, the overhead of information tracking is unaffected by this optimization, leaving a residual overhead, which can still be too much in certain applications. With the growing interest in data-only attacks, reducing the overhead required by these instrumentations will be the key in keeping one step ahead of malicious actors.

In this dissertation, we argue that the problems presented above can be mitigated or even eliminated in certain instances via careful problem understanding and robust design. We tackled each problem individually aiming to improve the quality to the end-to-end vulnerability detection and mitigation. We first analyzed the reasoning behind the difficulty to detect vulnerabilities using traditional software testing tools. Based on these lessons we propose a novel testing approach to detect complex memory corruption bugs hidden deep within the code. This approach also showcases the need to have a better understanding of the software under test and we propose a novel data structure detection scheme for this purpose. Our experience with testing allowed us to evaluate the control-flow integrity solution implemented in the GCC compiler, leading us to find design and implementation faults which undermine its purpose. We used this knowledge to generate the first practical ground-truth about control-flow specifications, allowing us to automatically test and evaluate the effectiveness of similar solutions in the future. Based on this data we also propose a novel

redesign of the system within the GCC compiler with provable optimality in terms of security. Lastly we explored the space of instrumentation techniques designed to discover faults during testing, to infer their requirements in terms of run-time information tracking. We found that a common weakness of many of these systems is the difficulty in associating allocations with custom run-time information efficiently. Because of this finding, we propose a novel information tracking scheme which can easily be plugged into these systems, while keeping run-time overhead at a minimum. By improving the performance of these instrumentation techniques, we hope to bring them closer to production systems, to allow them to mitigate against the recent developments in data-only attacks.

In the following we offer a short description about each of these topics and the contributions made within.

Guided testing for buffer overflows

In this dissertation, we first wanted to answer the research question if symbolic execution can be adapted to detect buffer overflow type bugs. These bugs occur in unsafe languages, such as C/C++, when an array dereference is corrupted to point to a location outside of the target array. This can happen when malicious actors trigger unexpected behavior within the offset computation logic. We selected symbolic execution as the software testing technique of interest, since it is the primary fully automatic technique used today both by industry and academia [53; 106; 21; 28]. Symbolic execution is named after its mechanisms to keep track of the relationship between the program variables and inputs during execution. Expressions in the code are modeled as explicit entities, instead of their results being computed using concrete values. Symbolic execution then tracks each branching statement in the code to infer the precise input values which trigger the execution of a certain code-path. The latter allows the technique to only generate program inputs which trigger a new code-path.

As mentioned earlier, memory corruption bugs are still highly prevalent [128; 125] and our evaluations in this dissertation show that this class of bugs is particularly difficult to detect using the symbolic execution tools available to researchers. We identified that the code-coverage heuristics in use by these systems are in direct contradiction with the characteristics of buffer overflows. Buffer overflows occur because array indices can be corrupted after a series of operations, which typically occur after deep within loops after a certain number of loop iterations (as the offset progresses slowly in every iteration). In contrast, code-coverage metrics push the testing system to exit loops as quickly as possible, as multiple iterations rarely increase the number of code artifacts covered by a given test input. Naively changing the heuristics to execute more loop iterations is not realistic, since the minimum number of iterations required to trigger the bug is not known a-priori and testing should still progress from one loop to another.

We decided to delve more deeply into the characteristics of exploitable buffer overflow bugs, to help us define a novel and efficient heuristic. After analyzing a many buffer overflow vulnerabilities reported in CVE (database of exploitable bugs [91]), we inferred that these bugs typically occur either due to a misuse of string processing functions or due to loops having complex index computation logic. Since the first category has been addressed by modern software development processes [32], we focused our attention to the second category. We found that 80% of the bugs within loops reside in the top 20% most complex loops in terms of offset computation logic. This suggests the need to treat different loops with different priorities during the testing process to maximize efficiency.

We use these finding to define a new gray-box testing mechanism, called guided fuzzing. Guided fuzzing uses code analysis to identify interesting code artifacts related to a specific testing goal and applies symbolic execution iteratively, with each iteration focused around a certain artifact. Since we operate in a gray-box fashion, information about software modularity is not available, thus we need alternative mechanisms to restrict symbolic execution to a certain artifact. We leverage taint analysis to infer implicit modularity within the program inputs, expecting that each interesting artifact is only influenced by a subset of the input space. By keeping unrelated program inputs fixed during guided fuzzing the search space of symbolic execution is restricted to code which can potentially influence the desired artifact. We found that in practice input modularity is not enough to scale guided fuzzing on complex programs, thus we proposed a second novel mechanism, based on machine learning, to further focus the search. We propose using small inputs to learn about the influences of individual branch statements and the target artifact and use this information to prioritize the most interesting branch statements when searching with a larger search-space.

The combination of guiding mechanism allowed us to quickly detect buffer overflow vulnerabilities in several complex loops within applications, where existing symbolic execution tools fail. While these results are specific to buffer overflows, we believe that guided fuzzing is a valid technique for any bug family, which can be associated to a certain type of code artifacts.

The guiding mechanisms were prototyped during my master thesis to check their technical feasibility on a singular webserver type application. During this dissertation we analyzed and confirmed the validity of guided fuzzing for the use case of buffer overflow detection. The guiding mechanisms were also updated and refined to deal with a proper selection of target programs.

Pointer-structure reversing

Our results with guided fuzzing showed great potential in using source analysis to improve testing efficiency, but software security sometimes needs to deal with closed-source binaries. One example of this scenario is the need to check third-party

components, which might affect the security of our software systems. We propose to replace the source-level analysis in the proposed guided fuzzing approach with recent advances in the field of binary reversing. For example, data structure analysis has gained a lot of ground recently, with systems [119; 85] showing great reliability and accuracy at identifying core data structures such as arrays or structs. One area we found lacking in these tools however, were pointer structures, data structures resulting from a connection of multiple heap objects, such as linked lists and trees. So we posed the following research question: Can one identify and classify pointer structures accurately within closed-source binaries?

We propose a novel system to identify and classify heap structures from an execution trace, without relying on other information sources or assumptions about the program under test. The classification operates by splitting the heap into disjoint sets of objects, according to course-grained typing inferred from the trace itself. The core intuition behind this typing mechanism is the idea that objects of the same functional type are operated on by the same group of instructions. In practice, we found that data structure shapes for the same functional type can vary wildly from one implementation to another as developers each have their own optimization tricks. Instead of complicating the shape inference logic, we decided on using a series of normalization steps instead. These steps attempt to uncover the underlying structure, based on the post-conditions guaranteed by the previous stages and observations made about real-world examples. During evaluation, the system showed great precision in dealing with a wide-range of implementations both from real-world binaries as well as specific data structure libraries, confirming the benefits of the proposed approach.

Verified control-flow integrity

2013 saw the resurgence of control-flow integrity [141; 139] originally published in 2005 [3]. These implementations sacrificed some of the precision described in the original paper to reduce the overhead and make the technique feasible to implement in practice. Unfortunately, the lack of any reference implementation meant that the security impact of the precision loss could never be evaluated objectively. Further refinements were made to CFI in the last couple of years, but the evaluation issues continued to persist. With our experience in software testing we posed the question: Can we adapt some software testing technique to objectively evaluate the attack surface remaining after control-flow integrity has been deployed?

We choose to target a specific sub-class of control-flow integrity our evaluation, namely the protection of vtable-based call-sites. These code artifacts are specific to the C++ dynamic dispatch mechanism and Google has shown them to be the main attack surface in their browser [127]. What makes these artifacts special for CFI is the availability of high-level C++ semantics which describe the rules for optimal protection. C++ defines that vtable-based call-sites can only ever call into polymorphic variants of the method described at the call-site. Several papers [127; 72] apply

this heuristic to claim optimal protection for this class of call-sites, but none of them offer a comprehensive validation of their implementation.

We propose a novel white-box testing framework to test the behavior of these implementations across all possible class hierarchies of a certain size. White-box testing describes a manually designed testing suite aimed to test a certain program feature comprehensively. Unfortunately, compilers are still too complex to be tested comprehensively by automatic mechanisms, such as symbolic execution. Out of all the vtable-based control-flow integrity solutions, only the one presented by Tice et.al, VTV [127], has been made public, thus we were unable to test the other implementations. To compensate for this. we made the framework open-source and contacted the other authors about its availability for future evaluation. To our surprise, we found multiple implementation and design issues in VTV already. This was unexpected, considering that it has been available in GCC since version 4.9.0 (released more than a year before our experiments).

We also used our experiments to drive the design of a new vtable-protection scheme based on VTV. The new system was also rigorously tested with the framework presented above to ensure optimal protection for vtable-based call-sites, as well as full application compatibility (modulo the correctness of the testing frame-work). Not only does the new design offer provably more security than the official version of VTV, it also reduces its impact on performance. Typically, security comes at a performance cost, making our solution exceptionally effective, beating out a production feature in all characteristics. These results show that comprehensive testing of CFI is not impossible and this field should be explored further to eliminate potential implementation and design bugs.

Efficient and uniform metadata tracking

As data-only attacks become more prominent with each passing conference [23; 70], it becomes obvious that run-time verification is necessary to complement programs written in unsafe languages. While several instrumentation techniques [115; 137; 84; 7; 6] have been shown to effectively stop data-only attacks, none of them made it into production systems, due to the overhead they introduce. As mentioned earlier, the overhead introduced by the verification itself can be scaled to the individual requirements, but these systems share a common source of fixed overhead from the need to track meta-information about memory objects. As such, we posed the research question: “Is it possible to track complex metadata for objects residing in all types of memory with low memory and run-time overhead?”.

To answer this question, we propose a novel metadata tracking scheme based around the concept of object alignment. It is based on a simple observation, that if all objects within a certain memory page are aligned to a certain byte-count, then it is enough to store metadata only once for every alignment-sized slot. This is due to the inherent property that no slot is shared between two objects. The slot identifier can

also easily be computed for any pointer, just by knowing the alignment associated to the corresponding memory page. This allows quick retrieval of the metadata with the help of page-specific information, which can be stored in a page-table-like data structure.

Organizing memory to correspond to the alignment restriction traditionally requires custom-memory allocators and a custom stack layouts, which may break application compatibility and performance. Fortunately, we discovered that some recent memory management techniques already offer a suitable building-block to support our scheme and we could leverage their inherent performance and compatibility guarantees to our benefit. The tcmalloc allocator [49] used by the Chrome browser (and other Google projects) groups memory objects of similar sizes into each page to speed up free-list tracking. This maps well to our requirement of having uniform alignments within all memory pages on the heap. In case of the stack we benefit from the recent developments in shadow-stack organization within the LLVM compiler, namely SafeStack [80], to move objects subjected to instrumentation onto a secondary custom stack, where no ABI restrictions are in place.

By building on top of such strong existing systems, our proposed metadata tracking preserves the application compatibility that they offer as well as their fine-tuned performance. In our experiments the addition of metadata tracking resulted in a minimal overhead on top of the base system, even when tracking multiple bytes of additional information for each object. This result enables the progressive deployment of run-time instrumentation as foreshadowed by other researchers [131].

To validate our approach, we used it as a building block to tackled a key emerging topic in unsafe languages, namely type safety. The existing solutions in this space [105; 29; 84] were limited in scope due to the difficulty of tracking type information along with objects allocated in different memory regions (stack, heap, etc.). Our experience from this experiment was that the metadata tracking approach was easy to integrate with the requirements of type tracking and verification, while easily beating out previous solutions in terms of performance. The tracking itself ended up being so efficient, that the overhead could be scaled all the way down to a 5% margin on relevant applications by sacrificing coverage, similar to the approach of Wagner et al. [131].

Organization of the Dissertation

This dissertation makes several contributions, with results published in refereed conferences and workshops (Page xvii). The remainder is organized as follows:

- Chapter 2 presents DOWSER, a guided fuzzer using source analysis to identify potentially dangerous loops in the program to focus symbolic execution around said code fragments. Its guidance mechanism includes taint analysis and learning to minimize the cost required to test of the specified code fragments for buffer overflow type bugs. This guidance and the focus on the individual code arti-

facts allows DOWSER to detect complex bugs deep within medium-sized applications, which was shown to be impossible with the existing systems. Chapter 2 appeared in *Proceedings of the 22nd USENIX Security Symposium (Usenix Security 13)* [62]. Personal contributions to this paper include the entire technical background (design, development and evaluation) as well as the development of the guided fuzzing concept.

- Chapter 3 presents MEMPICK, a binary analysis framework, designed to detect and classify pointer-based data structures, such as linked list or trees. It can accurately classify said data structures without any information from the binary, such as type information or functional boundaries. We demonstrate that it is possible to perform accurate classification on a wide-range of data structure shapes and implementations with these restrictions in mind, by using a simple set of normalization heuristics. Chapter 3 appeared in the journal *Empirical Software Engineering 2015* [64]. Personal contributions to this paper include the entire technical background (design, development and evaluation).

- Chapter 4 presents SHRINKWRAP, a set of design guidelines to implement vtable-based control-flow integrity correctly to correspond with the C++ semantics. It is based on the evaluation of existing solutions, which are shown to feature multiple design and implementation bugs, when subjected to the rigorous evaluation within SHRINKWRAP. We also apply the design guidelines the protection implemented in the GCC compiler today showcasing performance and security improvements at the same time, while preserving application compatibility. SHRINKWRAP is the first example of a provably correct control-flow integrity implementation and its evaluation framework can also be used by other implementations to validate their correctness. Chapter 4 appeared in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC 2015)* [63]. Personal contributions to this paper include the design and implementation of the evaluation framework on one hand and the concepts within the design guidelines as well as the development of the SHRINKWRAP prototype on the other hand.

- Chapter 5 presents METALLOC a framework designed to efficiently track object metadata at run-time. It is designed with flexibility in mind, allowing it to support a wide-range of instrumentation techniques, including bounds-checking, type-based validation, data integrity or dangling pointer detection. METALLOC is designed to follow recent developments in memory organization, thus it can take advantage of and existing high-performance memory allocator as well as custom stack layouts allowed within the compiler. Chapter 5 appeared in *Proceedings of the 9th European Workshop on System Security (EuroSec 2016)* [66]. Personal contributions to this paper include the entire technical background (design, development and evaluation) as well as the development of the variable alignment memory shadowing concept.

- Chapter 6 presents TYPESAN a compiler-based instrumentation scheme which

validate potentially unsafe down-casts from C++ programs at run-time. Internally is uses METALLOC to track the type information internally at a very low cost, validating our expectations about its flexibility as well as performance profile. TYPESAN showcases the ability to tackle the problem of type safety with a low performance overhead, while also significantly improving coverage across type casting artefacts. Chapter 6 appeared in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* [65]. Personal contributions to this paper include the design and development of all concepts related to the paper.

- Chapter 7 concludes the dissertation, summarizing key results, analyzing current limitations, and highlighting opportunities for future research directions.

2

Dowsing for overflows: A guided fuzzer to find buffer boundary violations

Abstract

DOWSER is a 'guided' fuzzer that combines taint tracking, program analysis and symbolic execution to find buffer overflow and underflow vulnerabilities buried deep in a program's logic. The key idea is that analysis of a program lets us pinpoint the right areas in the program code to probe and the appropriate inputs to do so.

Intuitively, for typical buffer overflows, we need consider only the code that accesses an array in a loop, rather than all possible instructions in the program. After finding all such candidate sets of instructions, we rank them according to an estimation of how likely they are to contain interesting vulnerabilities. We then subject the most promising sets to further testing. Specifically, we first use taint analysis to determine which input bytes influence the array index and then execute the program symbolically, making only this set of inputs symbolic. By constantly steering the symbolic execution along branch outcomes most likely to lead to overflows, we were able to detect deep bugs in real programs (like the `nginx` webserver, the `inspircd` IRC server, and the `ffmpeg` videoplayer). Two of the bugs we found were previously undocumented buffer overflows in `ffmpeg` and the `poppler` PDF rendering library.

2.1 Introduction

We discuss DOWSER, a 'guided' fuzzer that combines taint tracking, program analysis and symbolic execution, to find buffer overflow bugs buried deep in the program's logic.

Buffer overflows are perennially in the top 3 most dangerous software errors [36] and recent studies suggest this will not change any time soon [128; 122]. There are two ways to handle them. Either we harden the software with memory protectors that terminate the program when an overflow occurs (at runtime), or we track down the vulnerabilities before releasing the software (e.g., in the testing phase).

Memory protectors include common solutions like shadow stacks and canaries [33], and more elaborate compiler extensions like WIT [6]. They are effective in preventing programs from being exploited, but they do not remove the overflow bugs themselves. Although it is better to crash than to allow exploitation, crashes are undesirable too!

Thus, vendors prefer to squash bugs beforehand and typically try to find as many as they can by means of fuzz testing. Fuzzers feed programs invalid, unexpected, or random data to see if they crash or exhibit unexpected behavior[1]. As an example, Microsoft made fuzzing mandatory for every untrusted interface for every product, and their fuzzing solution has been running 24/7 since 2008 for a total of over 400 machine years [53].

Unfortunately, the effectiveness of most fuzzers is poor and the results rarely extend beyond shallow bugs. Most fuzzers take a 'blackbox' approach that focuses on the input format and ignores the tested software target. Blackbox fuzzing is popular and fast, but misses many relevant code paths and thus many bugs. Blackbox fuzzing is a bit like shooting in the dark: you have to be lucky to hit anything interesting.

Whitebox fuzzing, as implemented in [53; 21; 28], is more principled. By means of symbolic execution, it exercises all possible execution paths through the program and thus uncovers all possible bugs – although it may take years to do. Since full symbolic execution is slow and does not scale to large programs, it is hard to use it to find complex bugs in large programs [21; 28]. In practice, the aim is therefore to first cover as much unique code as possible. As a result, bugs that require a program to execute the same code many times (like buffer overflows) are hard to trigger except in very simple cases.

Eventual completeness, as provided by symbolic execution, is both a strength and a weakness, and in this paper, we evaluate the exact opposite strategy. Rather than testing all possible execution paths, we perform *spot checks* on a small number of code areas that look likely candidates for buffer overflow bugs and test each in turn.

The drawback of our approach is that we execute a symbolic run for each candidate code area—in an iterative fashion. Moreover, we can discover buffer overflows

[1] See `http://www.fuzzing.org/` for a collection of available fuzzers

only in the loops that we can exercise. On the other hand, by homing in on promising code areas directly, we speed up the search considerably, and manage to find complicated bugs in real programs that would be hard to find with most existing fuzzers.

Contributions The goal we set ourselves was to develop an efficient fuzzer that actively *searches* for buffer overflows *directly*. The key insight is that careful analysis of a program lets us pinpoint the right places to probe and the appropriate inputs to do so. The main contribution is that our fuzzer directly zooms in on these buffer overflow candidates and explores a novel 'spot-check' approach in symbolic execution.

To make the approach work, we need to address two main challenges. The first challenge is *where* to steer the execution of a program to increase the chances of finding a vulnerability. Whitebox fuzzers 'blindly' try to execute as much of the program as possible, in the hope of hitting a bug eventually. Instead, DOWSER uses information about the target program to identify code that is most likely to be vulnerable to a buffer overflow.

For instance, buffer overflows occur (mostly) in code that accesses an array in a loop. Thus, we look for such code and ignore most of the remaining instructions in the program. Furthermore, DOWSER performs static analysis of the program to *rank* such accesses. We will evaluate different ranking functions, but the best one so far ranks the array accesses according to complexity. The intuition is that code with convoluted pointer arithmetic and/or complex control flow is more prone to memory errors than straightforward array accesses. Moreover, by focusing on such code, DOWSER prioritizes bugs that are complicated—typically, the kind of vulnerabilities that static analysis or random fuzzing cannot find. The aim is to reduce the time wasted on shallow bugs that could also have been found using existing methods. Still, other rankings are possible also, and DOWSER is entirely agnostic to the ranking function used.

The second challenge we address is *how* to steer the execution of a program to these "interesting" code areas. As a baseline, we use *concolic* execution [134]: a combination of concrete and symbolic execution, where the concrete (fixed) input starts off the symbolic execution. In DOWSER, we enhance concolic execution with two optimizations.

First, we propose a new path selection algorithm. As we saw earlier, traditional symbolic execution aims at code coverage—maximizing the fraction of individual branches executed [21; 53]. In contrast, we aim for *pointer value* coverage of *selected* code fragments. When DOWSER examines an interesting pointer dereference, it steers the symbolic execution along branches that are likely to alter the value of the pointer.

Second, we reduce the amount of symbolic input as much as we can. Specifically, DOWSER uses dynamic taint analysis to determine which input bytes influence the

pointers used for array accesses. Later, it treats only these inputs as symbolic. While taint analysis itself is not new, we introduce novel optimizations to arrive at a set of symbolic inputs that is as *accurate* as possible (with neither too few, nor too many symbolic bytes).

In summary, DOWSER is a new fuzzer targeted at vendors who want to test their code for buffer overflows and underflows. We implemented the analyses of DOWSER as LLVM [81] passes, while the symbolic execution step employs S2E [28]. Finally, DOWSER is a *practical* solution. Rather than aiming for all possible security bugs, it specifically targets the class of buffer overflows (one of the most, if not the most, important class of attack vectors for code injection). So far, DOWSER found several real bugs in complex programs like `nginx`, `ffmpeg`, and `inspircd`. Most of them are extremely difficult to find with existing symbolic execution tools.

Assumptions and outline Throughout this paper, we assume that we have a test suite that allows us to reach the array accesses. Instructions that we cannot reach, we cannot test. In the remainder, we start with a big picture and the running example (Section 2.2). Then, we discuss the three main components of DOWSER in turn: the selection of interesting code fragments (Section 2.3), the use of dynamic taint analysis to determine which inputs influence the candidate instructions (Section 2.4), and our approach to nudge the program to trigger a bug during symbolic execution (Section 2.5). We evaluate the system in Section 3.10, discuss the related projects in Section 3.13. We conclude in Section 5.5.

2.2 Big picture

The main goal of DOWSER is to manipulate the pointers that instructions use to access an array in a loop, in the hope of forcing a buffer overrun or underrun.

2.2.1 Running example

Throughout the paper, we will use the function in Figure 2.1 to illustrate how dowsing works. The example is a simplified version of a buffer underrun vulnerability in the `nginx-0.6.32` web server [2]. A specially crafted input tricks the program into setting the `u` pointer to a location outside its buffer boundaries. When this pointer is later used to access memory, it allows attackers to overwrite a function pointer, and execute arbitrary programs on the system.

Figure 2.1 presents only an excerpt from the original function, which in reality spans approximately 400 lines of C code. It contains a number of additional options in the `switch` statement, and a few nested conditional `if` statements. This complexity severely impedes detecting the bug by both static analysis tools and symbolic execution engines. For instance, when we steered S2E [28] all the way down to the vulnerable function, and made solely the seven byte long uri path of the HTTP

A buffer underrun vulnerability in `nginx`

```
[1] int ngx_http_parse_complex_uri(ngx_http_request_t *r)
[2] {
[3]     state = sw_usual;
[4]     u_char* p = r->uri_start;   // user input
[5]     u_char* u = r->uri.data; // store normalized uri here
[6]     u_char ch = *p++;           // the current character
[7]
[8]     while (p <= r->uri_end) {
[9]         switch (state) {
[10]            case sw_usual:
[11]                if (ch == '/')
[12]                    state = sw_slash; *u++ = ch;
[13]                else /* many more options here */
[14]                ch = *p++; break;
[15]
[16]            case sw_slash:
[17]                if (ch == '/')
[18]                    *u++ = ch;
[19]                else if (ch == '.')
[20]                    state = sw_dot; *u++ = ch;
[21]                else /* many more options here */
[22]                ch = *p++; break;
[23]
[24]            case sw_dot:
[25]                if (ch == '.')
[26]                    state = sw_dot_dot; *u++ = ch;
[27]                else /* many more options here */
[28]                ch = *p++; break;
[29]
[30]            case sw_dot_dot:
[31]                if (ch == '/')
[32]                    state = sw_slash; u -=4;
[33]                    while (*(u-1) != '/') u--;
[34]                else /* many more options here */
[35]                ch = *p++; break;
[36]        }
[37]    }
[38] }
```

`Nginx` is a web server—in terms of market share across the million busiest sites, it ranks third in the world. At the time of writing, it hosts about 22 million domains worldwide. Versions prior to 0.6.38 had a particularly nasty vulnerability [2]. When `nginx` receives an HTTP request, the parsing function `nginx_http_parse_complex_uri`, first normalizes a uri path in `p=r->uri_start` (line 4), storing the result in a heap buffer pointed to by `u=r->uri.data` (line 5). The `while-switch` implements a state machine that consumes the input one character at a time, and transform it into a canonical form in `u`.

The source of the vulnerability is in the `sw_dot_dot` state. When provided with a carefully crafted path, `nginx` wrongly sets the beginning of `u` to a location somewhere below `r->uri.data`. Suppose the uri is `"//../foo"`. When `p` reaches `"/foo"`, `u` points to `(r->uri.data+4)`, and `state` is `sw_dot_dot` (line 30). The routine now decreases `u` by 4 (line 32), so that it points to `r->uri.data`. As long as the memory below `r->uri.data` does not contain the character `"/"`, `u` is further decreased (line 33), even though it crosses buffer boundaries. Finally, the user provided input (`"foo"`) is copied to the location pointed to by `u`.

In this case, the overwritten buffer contains a pointer to a function, which will be eventually called by `nginx`. Thus the vulnerability allows attackers to modify a function pointer, and execute an arbitrary program on the system.

It is a complex bug that is hard to find with existing solutions. The many conditional statements that depend on symbolic input are problematic for symbolic execution, while input-dependent indirect jumps are also a bad match for static analysis.

Figure 2.1: A simplified version of a buffer underrun vulnerability in `nginx`.

message symbolic, it took over 60 minutes to track down the problematic scenario. A more scalable solution is necessary in practice. Without these hints, S2E did not find the bug at all during an eight hour long execution.[2] In contrast, DOWSER finds it in less than 5 minutes.

The primary reason for the high cost of the analysis in S2E is the large number of conditional branches which depend on (symbolic) input. For each of the branches, symbolic execution first checks whether either the condition or its negation is satisfiable. When both branches are feasible, the default behavior is to examine both. This procedure results in an exponentially growing number of paths.

[2] All measurements in the paper use the same environment as in Section 3.10.

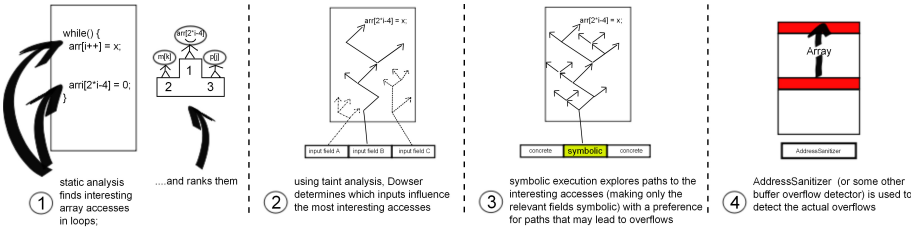


Figure 2.2: DOWSER– high-level overview.

This real-world example shows the need for (1) focusing the powerful yet expensive symbolic execution on the most interesting cases, (2) making informed branch choices, and (3) minimizing the amount of symbolic data.

2.2.2 High-level overview

Figure 2.2 illustrates the overall DOWSER architecture.

First, it performs a data flow analysis of the target program, and ranks all instructions that access buffers in loops ①. While we can rank them in different ways and DOWSER is agnostic as to the ranking function we use, our experience so far is that an estimation of complexity works best. Specifically, we rank calculations and conditions that are more complex higher than simple ones. In Figure 2.1, `u` is involved in three different operations, i.e., `u++`, `u-`, and `u-=4`, in multiple instructions inside a loop. As we shall see, these intricate computations place the dereferences of `u` in the top 3% of the most complex pointer accesses across `nginx`.

In the second step ②, DOWSER repeatedly picks high-ranking accesses, and selects test inputs which exercise them. Then, it uses dynamic taint analysis to determine which input bytes influence pointers dereferenced in the candidate instructions. The idea is that, given the format of the input, DOWSER fuzzes (i.e., treats as symbolic), only those fields that affect the potentially vulnerable memory accesses, and keeps the remaining ones unchanged. In Figure 2.1, we learn that it is sufficient to treat the uri path in the HTTP request as symbolic. Indeed, the computations inside the vulnerable function are independent of the remaining part of the input message.

Next ③, for each candidate instruction and the input bytes involved in calculating the array pointer, DOWSER uses symbolic execution to try to nudge the program toward overflowing the buffer. Specifically, we execute symbolically the loop that contains the candidate instructions (and thus should be tested for buffer overflows)—treating only the relevant bytes as symbolic. As we shall see, a new path selection algorithm helps to guide execution to a possible overflow quickly.

Finally, we detect any overflow that may occur. Just like in whitebox fuzzers, we can use any technique to do so (e.g., Purify, Valgrind [96], or BinArmor [120]). In our work, we use Google's AddressSanitizer [115] ④. It instruments the protected program to ensure that memory access instructions never read or write so called,

"poisoned" red zones. Red zones are small regions of memory inserted inbetween any two stack, heap or global objects. Since they should never be addressed by the program, an access to them indicates an illegal behavior. This policy detects sequential buffer over- and underflows, and some of the more sophisticated pointer corruption bugs. This technique is beneficial when searching for new bugs since it will also trigger on silent failures, not just application crashes. In the case of `nginx`, AddressSanitizer detects the underflow when the `u` pointer reads memory outside its buffer boundaries (line 33).

We explain step ① (static analysis) in Section 2.3, step ② (taint analysis) in Section 2.4, and step ③ (guided execution) in Section 2.5.

2.3 Dowsing for candidate instructions

Previous research has shown that software complexity metrics collected from software artifacts are helpful in finding vulnerable code components [48; 142; 117; 98]. However, even though complexity metrics serve as useful indicators, they also suffer from low precision or recall values. Moreover, most of the current approaches operate at the granularity of modules or files, which is too coarse for the directed symbolic execution in DOWSER.

As observed by Zimmermann et al. [142], we need metrics that exploit the unique characteristics of vulnerabilities, e.g., buffer overflows or integer overruns. In principle, DOWSER can work with any metric capable of ranking groups of instructions that access buffers in a loop. So, the question is how to design a good metric for complexity that satisfies this criterion? In the remainder of this section, we introduce one such metric: a heuristics-based approach that we specifically designed for the detection of potential buffer overflow vulnerabilities.

We leverage a primary pragmatic reason behind complex buffer overflows: convoluted pointer computations are hard to follow by a programmer. Thus, we focus on 'complex' array accesses realized inside loops. Further, we limit the analysis to pointers which evolve together with loop induction variables, i.e., are repeatedly updated to access (various) elements of an array.

Using this metric, DOWSER ranks buffer accesses by evaluating the complexity of data- and control-flows involved with the array index (pointer) calculations. For each loop in the program, it first statically determines (1) the set of all instructions involved in modifying an array pointer (we will call this a pointer's *analysis group*), and (2) the conditions that guard this analysis group, e.g., the condition of an `if` or `while` statement containing the array index calculations. Next, it labels all such sets with scores reflecting their complexity. We explain these steps in detail in Sections 2.3.1, 2.3.2, and 2.3.3.

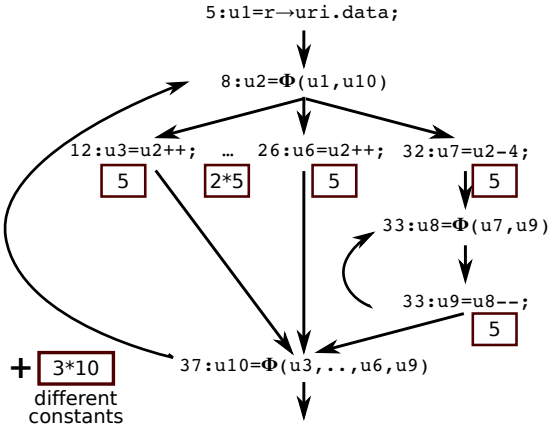


Figure 2.3: Data-flow graph and analysis group associated with the pointer `u` from Figure 2.1. For the sake of clarity, the figure presents pointer arithmetic instructions in pseudo code. The PHI nodes represent locations where data is merged from different control-flows. The numbers in the boxes represent points assigned by DOWSER.

2.3.1 Building analysis groups

Suppose a pointer `p` is involved in an "interesting" array access instruction $\texttt{acc}_\texttt{p}$ in a loop. The *analysis group* associated with $\texttt{acc}_\texttt{p}$, $\texttt{AG}(\texttt{acc}_\texttt{p})$, collects all instructions that influence the value of the dereferenced pointer during the execution of the loop.

To determine $\texttt{AG}(\texttt{acc}_\texttt{p})$, we compute an intraprocedural data flow graph representing operations in the loop that compute the value of `p` dereferenced in $\texttt{acc}_\texttt{p}$. Then, we check if the graph contains cycles. A cycle indicates that the value of `p` in a previous loop iteration affects its value in the current one, so `p` depends on the loop induction variable.

As mentioned before, this part of our work is built on top of the LLVM [81] compiler infrastructure. The static single assignment (SSA) form provided by LLVM translates directly to data flow graphs. Figure 2.3 shows an example. Observe that, since all dereferences of pointer `u` share their data flow graph, they also form a single analysis group. Thus, when DOWSER later tries to find an illegal array access within this analysis group, it tests all the dereferences at the same time—there is no need to consider them separately.

2.3.2 Conditions guarding analysis groups

It may happen that the data flow associated with an array pointer is simple, but the value of the pointer is hard to follow due to some complex control changes. For this reason, DOWSER ranks also control flows: the conditions that influence an analysis group.

Say that an instruction manipulating the array pointer `p` is guarded by a condition

on a variable `var`, e.g., `if(var<10){*p++=0;}`. If the value of `var` is difficult to keep track of, so is the value of `p`. To assess the complexity of `var`, DOWSER analyzes its data flow, and determines the analysis group, `AG(var)` (as discussed in Section 2.3.1). Moreover, we recursively analyze the analysis groups of other variables influencing `var` and `p` inside the loop. Thus, we obtain a number of analysis groups which we rank in the next step (Section 2.3.3).

2.3.3 Scoring array accesses

For each array access realized in a loop, DOWSER assesses the complexity of the analysis groups constructed in Sections 2.3.1 and 2.3.2. For each analysis group, it considers all instructions, and assigns them points. The more points an AG cumulatively scores, the more complex it is. The overall rank of the array access is determined by the maximum of the scores. Intuitively, it reflects the most complex component.

The scoring algorithm should provide roughly the same results for semantically identical code. For this reason, we enforce the optimizations present in the LLVM compiler (e.g., to eliminate common subexpressions). This way, we minimize the differences in (the amount of) instructions arising from the compiler options. Moreover, we analyzed the LLVM code generation strategies, and defined a powerful set of *equivalence rules*, which minimize the variation in the scores assigned to syntactically different but semantically equivalent code. We highlight them below.

Table 2.1 introduces all types of instructions, and discusses their impact on the final score. In principle, all common instructions involved in array index calculations are of the order of 10 points, except for the two instructions that we consider risky: pointer casts and functions that return non-pointer values used in pointer calculation.

The absolute penalty for each type of instruction is not very important. However, we ensure that the points reflect the difference in complexity between various code fragments, instead of giving all array accesses the same score. That is, instructions that complicate the array index contribute to the score, and instructions that complicate the index a lot also score very high, relative to other instructions. In Section 3.10, we compare our complexity ranking to alternatives.

2.4 Using tainting to find inputs that matter

Once DOWSER has ranked array accesses in loops in order of complexity, we examine them in turn. Typically, only a small segment of the input affects the execution of a particular analysis group, so we want to search for a bug by modifying solely this part of the input, while keeping the rest constant (refer to Section 2.5). In the current section, we explain how DOWSER identifies the link between the components of the program input and the different analysis groups. Observe that this result also benefits other bug finding tools based on fuzzing, not just DOWSER and concolic execution.

Instructions	Rationale/Equivalence rules	Points
Array index manipulations		
Basic index arithmetic instr., i.e., addition and subtraction	`GetElemPtr`, that increases or decreases a pointer by an index, scores the same. Thus, operations on pointers are equivalent to operations on offsets. An instruction scores 1 if it modifies a value which is not passed to the next loop iteration.	1 or 5
Other index arithmetic instr. e.g., division, shift, or xor	These instructions involve more complex pointer calculations than the standard `add` or `sub`. Thus, we penalize them more.	10
Different constant values	Multiple constants used to modify a pointer make its value hard to follow. It is easier to keep track of a pointer that always increases by the same value.	10 per value
Constants used to access fields of structures	We assume that compilers handle accesses to structures correctly. We only consider constants used to compute the index of an array, and not the address of a field.	0
Numerical values determined outside the loop	Though in the context of the loop they are just constants, the compiler cannot predict their values. Thus they are difficult to reason about and more error prone.	30
Non-inlined functions returning non-pointer values	Since decoupling the computation of a pointer from its use might easily lead to mistakes, we heavily penalize this operation.	500
Data movement instructions	Moving (scalar or pointer) data does not add to the complexity of computations.	0
Pointer manipulations		
Load a pointer calculated outside the loop	It denotes retrieving the base pointer of an object, or using memory allocators. We treat all *remote* pointers in the same way - all score 0.	0
`GetElemPtr`	An LLVM instruction that computes a pointer from a base and offset(s). (See `add`.)	1 or 5
Pointer cast operations	Since the casting instructions often indicate operations that are not equivalent to the standard pointer manipulations (listed above), they are worth a close inspection.	100

Table 2.1: Overview of the instructions involved in pointer arithmetic operations, and their penalty points.

We focus our discussion on an analysis group $\texttt{AG(acc}_\texttt{p}\texttt{)}$ associated with an array pointer dereference $\texttt{acc}_\texttt{p}$. We assume that we can obtain a test input `I` that exercises the potentially vulnerable analysis group. While this may not always be true, we believe it is a reasonable assumption. Most vendors have test suites to test their software and they often contain at least one input which exercises each *complex* loop.

2.4.1 Baseline: dynamic taint analysis

As a basic approach, DOWSER performs dynamic taint analysis (DTA) [97] on the input `I` (tainting each input byte with a unique color, and propagating the colors on data movement and arithmetic operations). Then, it logs all colors and input bytes involved in the instructions in $\texttt{AG(acc}_\texttt{p}\texttt{)}$. Given the format of the input, DOWSER maps these bytes to individual fields. In Figure 2.1, DOWSER finds out that it is sufficient to treat `uri` as symbolic.

The problem with DTA, as sketched above, is that it misses *implicit flows* (also called *control dependencies*) entirely [45; 77]. Such flows have no direct assignment of a tainted value to a variable—which would be propagated by DTA. Instead, the value of a variable is completely determined by the value of a tainted variable in a condition. In Figure 2.1, even though the value of `u` in line 12 is dependent on the tainted character `ch` in line 11, the taint does not flow directly to `u`, so DTA would not report the dependency. Implicit flows are notoriously hard to track [118; 24], but ignoring them completely reduces our accuracy. DOWSER therefore employs a solution that builds on the work by Bao et al. [16], but with a novel optimization to increase the accuracy of the analysis (Section 2.4.2).

Like Bao et al. [16], DOWSER implements *strict control dependencies*. Intuitively, we propagate colors only on the most informative (or, information preserving) dependencies. Specifically, we require a direct comparison between a tainted variable and a compile time constant. For example, in Figure 2.1, we propagate the color of `ch` in line 11 to the variables `state` and `u` in line 12. However, we would keep `state` and `u` untainted if the condition in line 11 for instance had been either `"if(ch!='/')"` or `"if(ch<'/')"`. As implicit flows are not the focus of this paper we refer interested readers to [16] for details.

2.4.2 Field shifting to weed out false dependencies

Improving on the handling of strict control dependencies by Bao et al. [16], described above, DOWSER adds a novel technique to prevent overtainting due to false dependencies. The problems arise when the order of fields in an input format is not fixed, e.g., as in HTTP, SMTP (and the commandline for most programs). The approach from [16] may falsely suggest that a field is dependent on all fields that were extracted so far.

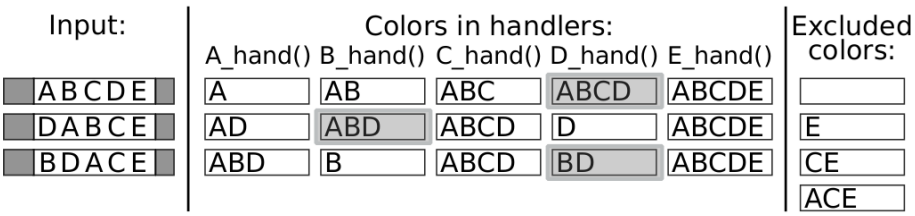


Figure 2.4: The figure shows how DOWSER shuffles an input to determine which fields really influence an analysis group. Suppose a parser extracts fields of the input one by one, and the analysis group depends on the fields B and D (with colors B and D, respectively). *Colors in handlers* show on which fields the subsequent handlers are strictly dependent [16], and the shaded rectangle indicates the colors propagated to the analysis group. *Excluded colors* are left out of our analysis.

For instance, `lighttpd` reads new header fields in a loop and compares them to various options, roughly as follows:

```
while () {
  if(cmp(field, "Content") == 0)
  ...
  else if(cmp(field, "Range") == 0)
  ...
  else exit (-1);
  field = extract_new_header_field();
}
```

As the parser tests for equivalence, the implicit flow will propagate from one field to the next one, even if there is no real dependency at all! Eventually, the last field appears to depend on the whole header.

DOWSER determines which options really matter for the instructions in an analysis group by *shifting* the fields whose order is not fixed. Refer to Figure 2.4, and suppose we have run the program with options A, B, C, D, and E, and our analysis group really depends on B and D. Once the message gets processed, we see that the AG does not depend on E, so E can be excluded from further analysis. Since the last observed color, D, has a direct influence on the AG, it is a true dependence. By performing a circular shift and re-trying with the order D, A, B, C, E, DOWSER finds only the colors corresponding to A, B, D. Thus, we can leave C out of our analysis. After the next circular shift, DOWSER reduces the colors to B and D only.

The optimization is based on two observations: (1) the last field propagated to the AG has a direct influence on the AG, so it needs to be kept, (2) all fields beyond this one are guaranteed to have no impact on the AG. By performing circular shifts, and running DTA on the updated input, DOWSER drops the undue dependencies.

Even though this optimization requires some minimal knowledge of the input, we do not need full understanding of the input grammar, like the contents or effects of fields. It is sufficient to identify the fields whose order is not fixed. Fortunately,

such information is available for many applications—especially when vendors test their own code.

2.5 Exploring candidate instructions

Once we have learnt which part of the program input influences the analysis group $\texttt{AG(acc}_\texttt{p}\texttt{)}$, we fuzz this part, and we try to nudge the program toward using the pointer `p` in an illegal way. More technically, we treat the interesting component of the input as symbolic, the remaining part as fixed (concrete), and we execute the loop associated with $\texttt{AG(acc}_\texttt{p}\texttt{)}$ symbolically.

However, since in principle the cost of a complete loop traversal is exponential, loops present one of the hardest problems for symbolic execution [51]. Therefore, when analyzing a loop, we try to select those paths that are most promising in our context. Specifically, DOWSER prioritizes paths that show a potential for knotty pointer arithmetic. As we show in Section 3.10, our technique significantly optimizes the search for an overflow.

DOWSER's loop exploration procedure has two main phases: learning, and bug finding. In the *learning phase*, DOWSER assigns each branch in the loop a weight approximating the probability that a path following this direction contains new pointer dereferences. The weights are based on statistics on the variety of pointer values observed during an execution of a short symbolic input.

Next, in the *bug finding phase*, DOWSER uses the weights determined in the first step to filter our uninteresting parts of the loop, and prioritize the important paths. Whenever the weight associated with a certain branch is 0, DOWSER does not even try to explore it further. In the vulnerable `nginx` parsing loop from which Figure 2.1 shows an excerpt, only 19 out of 60 branches scored a non-zero value, so were considered for the execution. In this phase, the symbolic input represents a real-world scenario, so it is relatively long. Therefore, it would be prohibitively expensive to be analyzed using a popular symbolic execution tool.

In Section 2.5.1, we briefly review the general concept of concolic execution, and then we discuss the two phases in Sections 2.5.2 and 2.5.3, respectively.

2.5.1 Baseline: concrete + symbolic execution

Like DART and SAGE [52; 53], DOWSER generates new test inputs by combining concrete and symbolic execution. This technique is known as *concolic* execution [114]. It runs the program on a concrete input, while gathering symbolic constraints from conditional statements encountered along the way. To test alternative paths, it systematically negates the collected constraints, and checks whether the new set is satisfiable. If so, it yields a new input. To bootstrap the procedure, DOWSER takes a test input which exercises the analysis group $\texttt{AG(acc}_\texttt{p}\texttt{)}$.

As mentioned already, a challenge in applying this approach is how to select the paths to explore first. The classic solution is to use depth first exploration of the paths

by backtracking [78]. However, since doing so results in an exponentially growing number of paths to be tested, the research community has proposed various heuristics to steer the execution toward unexplored regions. We discuss these techniques in Section 3.13.

2.5.2 Phase 1: learning

The aim of the learning phase is to rate the `true` and `false` directions of all conditional branches that depend on the symbolic input in the loop `L`. For each branch, we evaluate the likelihood that a particular outcome will lead to unique pointer dereferences (i.e., dereferences that we do not expect to find in the alternative outcome). Thus, we answer the question of how much we expect to gain when we follow this path, rather than the alternative. We encode this information into *weights*.

Specifically, the weights represent the likelihood of unique *access patterns*. An access pattern of the pointer `p` is the sequence of all values of `p` dereferenced during the execution of the loop. In Figure 2.1, when we denote the initial value of `u` by u_0, then the input `"//../"` triggers the following access pattern of the pointer `u`: (u_0, u_0+1, $u_0 + 2$, u_0-2,...).

To compute the weights, we learn about the effects of individual branches. In principle, each of them may (a) directly affect the value of a pointer, (b) be a precondition for another important branch, or (c) be irrelevant from the computation's standpoint. To distinguish between these cases, DOWSER analyzes all possible executions of a *short* symbolic input. By comparing the sets of `p`'s access patterns observed for both outcomes of a branch, it discovers which branches do not influence the diversity of pointer dereferences (i.e., are irrelevant).

Symbolic input In Section 2.4, we identified which part of the test input `I` we need to make symbolic. We denote this by $\mathtt{I}_S$. In the learning phase, DOWSER executes the loop `L` exhaustively. For performance reasons, we therefore further limit the amount of symbolic data and make only a short fragment of $\mathtt{I}_S$ symbolic. For instance, for Figure 2.1, the learning phase makes only the first 4 bytes of `uri` symbolic (not enough to trigger the bug), while scaling up to 50 symbolic bytes in the bug finding phase.

Algorithm DOWSER exhaustively executes `L` on a short symbolic input, and records how the decisions taken at conditional branch statements influence pointer dereference instructions. For each branch `b` along the execution path, we retain the access pattern of `p` realized during this execution, `AP(p)`. We informally interpret it as "if you choose the `true` (respectively, `false`) direction of the branch `b`, expect access pattern `AP(p)` (respectively, $\mathtt{AP}'(\mathtt{p})$)". This procedure results in two sets of access patterns for each branch statement, for the taken and non-taken branch, respectively. The final weight of each direction is the fraction of the access patterns that were unique for the direction in question, i.e., were not observed when the opposite one was taken.

The above description explains the intuition behind the learning mechanism, but the full algorithm is more complicated. The problem is that a conditional branch `b` might be exercised multiple times in an execution path, and it is possible that all the instances of `b` influence the access pattern observed.

Intuitively, to allow for it, we do not associate access patterns with just a single decision taken on `b` (`true` or `false`). Rather, each time `b` is exercised, we also retain which directions were previously chosen for `b`. Thus, we still collect "expected" access patterns if the `true` (respectively, `false`) direction of `b` is followed, but we augment them with a precondition. This way, when we compare the `true` and `false` sets to determine the weights for `b`, we base the scores on a deeper understanding of how an access pattern was reached.

Discussion It is important for our algorithm to avoid false negatives: we should not incorrectly flag a branch as irrelevant—it would preclude it from being explored in the bug finding phase. Say that `instr` is an instruction that dereferences the pointer `p`. To learn that a branch directly influences `instr`, it suffices to execute it. Similarly, since branches retain full access patterns of `p`, the information about `instr` being executed is also "propagated" to all its preconditions. Thus, to completely avoid false negatives, the algorithm would require full coverage of the instructions in an analysis group. We stress that we need to exercise all instructions, and not all paths in a loop. As observed by [21], exhaustive executions of even short symbolic inputs provide excellent instruction coverage in practice.

While false positives are undesirable as well, they only cause DOWSER to execute more paths in the second phase than absolutely necessary. Due to the limited path coverage, there are corner cases, when false positives can happen. Even so, in `nginx`, only 19 out of 60 branches scored a non-zero value, which let us execute the complex loop with a 50-byte-long symbolic input.

2.5.3 Phase 2: hunting bugs

In this step, DOWSER executes symbolically a real-world sized input in the hope of finding a value that triggers a bug. DOWSER uses the feedback from the learning phase (Section 2.5.2) to steer its symbolic execution toward new and interesting pointer dereferences. The goal of our heuristic is to avoid execution paths that do not bring any new pointer manipulation instructions. Thus, DOWSER shifts the target of symbolic execution from traditional *code* coverage to *pointer value* coverage.

DOWSER's strategy is explicitly dictated by the weights. As a baseline, the execution follows a depth-first exploration, and when DOWSER is about to select the direction of a branch `b` that depends on the symbolic input, it adheres to the following rules:

- If both the `true` and `false` directions of `b` have weight 0, we do not expect `b` to influence the variety of access patterns. Thus, DOWSER chooses the direction randomly, and does not intend to examine the other direction.

- If only one direction has a non-zero weight, we expect to observe unique access patterns only when the execution paths follows this direction, and DOWSER favors it.
- If both of `b`'s directions have non-zero weights, both the `true` and `false` options may bring unique access patterns. DOWSER examines both directions, and schedules them in order of their weights.

Intuitively, DOWSER's symbolic execution tries to select paths that are more likely to lead to overflows.

Guided fuzzing This concludes our description of DOWSER's architecture. To summarize, DOWSER helps fuzzing by: (1) finding "interesting" array accesses, (2) identifying the inputs that influence the accesses, and (3) fuzzing intelligently to cover the array. Moreover, the targeted selection procedure based on pointer value coverage and the small number of symbolic input values allow DOWSER to find bugs quickly and scale to larger applications. In addition, the ranking of array accesses permits us to zoom in on more complicated array accesses.

2.6 Evaluation

In this section, we first zoom in on the running example of `nginx` from Figure 2.1 to evaluate individual components of the system in detail (Section 2.6.1). In Section 2.6.2, we consider seven real-world applications. Based on their vulnerabilities, we evaluate our dowsing mechanism. Finally, we present an overview of the attacks detected by DOWSER.

Since DOWSER uses a 'spot-check' rather than 'code coverage' approach to bug detection, it must analyze each complex analysis group separately, starting with the highest ranking one, followed by the second one, and so on. Each of them runs until it finds a bug or gets terminated. The question is when we should terminate a symbolic execution run. Since symbolic execution of a single loop is highly optimized in DOWSER, we found each bug in less than 11 minutes, so we execute each symbolic run for a maximum of 15 minutes.

Our test platform is a Linux 3.1 system with an Intel(R) Core(TM) i7 CPU clocked at 2.7GHz with 4096KB L2 cache. The system has 8GB of memory. For our experiments we used an OpenSUSE 12.1 install. We ran each test multiple times and present the median.

2.6.1 Case study: Nginx

In this section, we evaluate each of the main steps of our fuzzer by looking at our case study of `nginx` in detail.

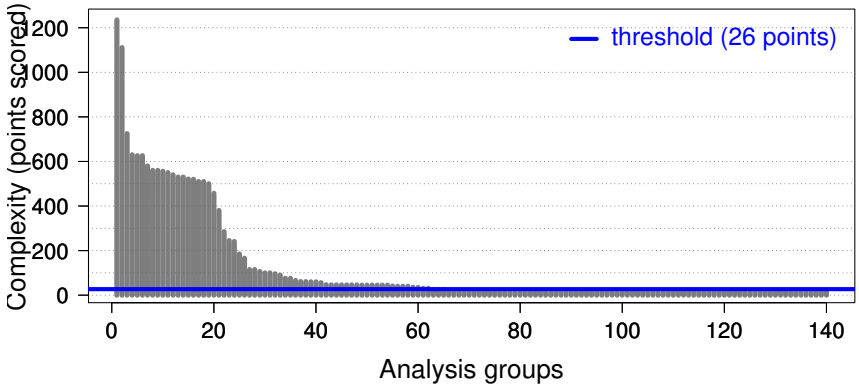


Figure 2.5: Scores of the analysis groups in `nginx`.

Dowsing for candidate instructions

We measure how well DOWSER highlights potentially faulty code and filters out the uninteresting fragments.

Our first question is whether we can filter out all the simple loops and focus on the more interesting ones. This turns out to be simple. Given the complexity scoring function from Section 2.3, we find that across all applications all analysis groups with a score less than 26 use just a single constant and at most two instructions modifying the offset of an array. Thus, in the remainder of our evaluation, we set our cut-off threshold to 26 points.

As shown in Table 2.2, `nginx` has 517 outermost loops, and only 140 analysis groups that access arrays. Thus, we throw out over 70% of the loops immediately[3]. Figure 2.5 presents the sorted weights of all the analysis groups in `nginx`. The distribution shows a quick drop after a few highly complex analysis groups. The long tail represents the numerous simple loops omnipresent in any code. 55.7% of the analysis groups score too low to be of interest. This means that DOWSER needs to examine only the remaining 44.3%, i.e., 62 out of 140 analysis groups, or at most 12% of all loops. Out of these, the buffer overflow in Figure 2.1 ranks 4th.

Taint analysis in context of hunting for bugs

In Section 2.4 we mentioned that 'traditional' dynamic taint analysis misses implicit flows, i.e., flows that have no direct assignment of a tainted value to a variable. The problem turns out to be particularly serious for `nginx`. It receives input in text format,

[3] In principle, if a loop accesses multiple arrays, it also contains multiple access groups. Thus, these 140 analysis groups are located in fewer than 140 loops.

and transforms it to extract numerical values or various flags. As such code employs conditional statements, DTA misses the dependencies between the input and analysis groups.

Next, we evaluate the usefulness of field shifting. First, we implement the taint propagation exactly as proposed by Bao et al. [16], without any further restrictions. In that case, an index variable in the `nginx` parser becomes tainted, and we mark all HTTP fields succeeding the `uri` field as tainted as well. As a result, we introduce more symbolic data than necessary. Next, we apply field shifting (Section 2.4.2) which effectively limits taint propagation to just the `uri` field. In general, the field shifting optimization improves the accuracy of taint propagation in all applications that take multiple input fields whose order does not matter. On the other hand, it will not help if the order is fixed.

Importance of guiding symbolic execution

We now use the `nginx` example to assess the importance of guiding symbolic execution to a vulnerability condition. For `nginx`, the input message is a generic HTTP request. Since it exercises the vulnerable loop for this analysis group, its `uri` starts with "//". Taint analysis allows us to detect that only the `uri` field is important, so we mark only this field as symbolic. As we shall see, without guidance, symbolic execution does not scale beyond very short `uri` fields (5-6 byte long). In contrast, DOWSER successfully executes 50-byte-long symbolic `uri`s.

When S2E [28] executes a loop, it can follow one of the two search strategies: depth-first search, or maximizing code coverage (as proposed in SAGE [53]). The first one aims at complete path coverage, and the second at executing basic blocks that were not seen before. However, none can be applied in practice to examine the complex loop in `nginx`. The search is so costly that we measured the runtime for only 5-6 byte long symbolic `uri` fields. The DFS strategy handled the 5-byte-long input in 139 seconds, the 6-byte-long in 824 seconds. A 7-byte input requires more than 1 hour to finish. Likewise, the code coverage strategy required 159, and 882 seconds, respectively. The code coverage heuristic does not speed up the search for buffer overflows either, since besides executing specific instructions from the loop, memory corruptions require a very particular execution context. Even if 100% code coverage is reached, they may stay undetected.

As we explained in Section 2.5, the strategy employed by DOWSER does not aim at full coverage. Instead, it actively searches for paths which involve new pointer dereferences. The learning phase uses a 4-byte-long symbolic input to observe access patterns in the loop. It follows a simple depth first search strategy. As the bug clearly cannot be triggered with this input size, the search continues in the second, hunting bugs, phase. The result of the learning phase disables 66% of the conditional branches significantly reducing the exponentially of the subsequent symbolic execution. Because of this heuristic, DOWSER easily scales up to 50 symbolic bytes and finds the bug after just a few minutes. A 5-byte-long symbolic input is handled in

20 seconds, 10 bytes in 42 seconds, 20 bytes in 63 seconds, 30 in 146 seconds, 40 in 174 seconds and 50 in 253 seconds. These numbers maintain an exponential growth of 1.1 for each added character. Even though DOWSER still exhibits the exponential behavior, the growth rate is fairly low. Even in the presence of 50 symbolic bytes, DOWSER quickly finds the complex bug.

In practice, symbolic execution has problems dealing with real world applications and input sizes. The number of execution paths quickly overwhelms these systems. Since triggering buffer overflows not only requires a vulnerable basic block, but also a special context, traditional symbolic execution tools are ill suited. DOWSER, instead, requires the application to be executed symbolically for only a very short input, and then it deals with real-world input sizes instead of being limited to a few input bytes. Combined with the ability to extract the relevant parts of the original input, this enables searching for bugs in applications like web servers where input sizes were considered until now to be well beyond the scalability of symbolic execution tools.

2.6.2 Overview

In this section, we consider several applications. First, we evaluate the dowsing mechanism, and we show that it successfully highlights vulnerable code fragments. Then, we summarize the memory corruptions detected by DOWSER. They come from six real world applications of several tens of thousands LoC, including the `ffmpeg` videoplayer of 300K LoC. The bug in `ffmpeg`, and one of the bugs in `poppler` were not documented before.

Dowsing for candidate instructions

We now examine several aspects of the dowsing mechanism. First, we show that there is a correlation between DOWSER scoring function and the existence of memory corruption vulnerabilities. Then, we discuss how our focus on complex loops limits the search space, i.e., the amount of analysis groups to be tested. We start with a description of our data set.

Data set To evaluate the effectiveness of DOWSER, we chose six real world programs: `nginx`, `ffmpeg`, `inspircd`, `libexif`, `poppler`, and `snort`. Additionally, we consider the vulnerabilities in `sendmail` tested by Zitser et al. [143]. For these applications, we analyzed all buffer overflows reported in CVE [91] since 2009. For `ffmpeg`, rather than include all possible codecs, we just picked the ones for which we had test cases. Out of 27 CVE reports, we took 17 for the evaluation. The remaining ten vulnerabilities are out of the scope of this paper – nine of them are related to an erroneous usage of a correct function, e.g., `strcpy`, and one was not in a loop. In this section, we consider the analysis groups from all the applications together, giving us

over 3000 samples, 17 of which are known to be vulnerable[4].

When evaluating DOWSER scoring mechanism, we also compare it to a straightforward scoring function that treats all instructions uniformly. For each array access, it considers exactly the same AGs as DOWSER. However, instead of the scoring algorithm (Table 2.1), each instruction gets 10 points. We will refer to this metric as `count`.

Correlation For both DOWSER and the `count` scoring functions, we computed the correlation between the number of points assigned to an analysis group and the existence of a memory corruption vulnerability. We used the Spearman rank correlation [4], since it is a reliable measure that is appropriate even when we do not know the probability distribution of the variables, or when the association between the variables is non-linear.

The positive correlation for DOWSER is statistically significant at $p < 0.0001$, for `count` — at $p < 0.005$. The correlation for DOWSER is stronger.

Dowsing The *Dowsing* columns of Table 2.2 shows that our focus on complex loops limits the search space from thousands of LoC to hundreds of loops, and finally to a small number of "interesting" analysis groups. Observe that `ffmpeg` has more analysis groups than loops. That is correct. If a loop accesses multiple arrays, it contains multiple analysis groups.

By limiting the analysis to complex cases, we focus on a smaller fraction of all AGs in the program, e.g., we consider 36.9% of all the analysis groups in `inspircd`, and 34.5% in `snort`. `ffmpeg`, on the other hand, contains lots of complex loops that decode videos, so we also observe many "complex" analysis groups.

In practice, symbolic execution, guided or not is expensive, and we can hardly afford a thorough analysis of more than just a small fraction of the target AGs of an application, say 20%-30%. For this reason, DOWSER uses a scoring function, and tests the analysis groups in order of decreasing score. Specifically, DOWSER looks at complexity. However, alternative heuristics are also possible. For instance, one may count the instructions that influence array accesses in an AG. To evaluate whether DOWSER's heuristics are useful, we compare how many bugs we discover if we examine increasing fractions of all AGs, in descending order of the score. So, we determine how many of the bugs we find if we explore the top 10% of all AGs, how many bugs we find when we explore the top 20%, and so on. In our evaluation, we are comparing the following ranking functions: (1) DOWSER's complexity metric, (2) counting instructions as described above, and (3) random.

Figure 2.6 illustrates the results. The random ranking serves as a baseline—clearly both `count` and DOWSER perform better. In order to detect all 17 bugs, DOWSER has to analyze 92.2% of all the analysis groups. However, even with just 15% of the targets, we find almost 80% (13/17) of all the bugs. At that same fraction of targets, `count` finds a little over 40% of the bugs (7/17). Overall, DOWSER

[4]Since the scoring functions are application agnostic, it is sound to compare their results across applications.

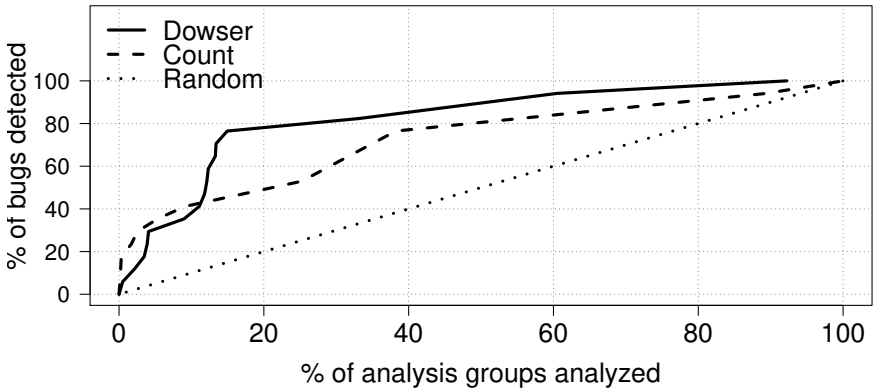


Figure 2.6: A comparison of random testing and two scoring functions: DOWSER and `count`. It illustrates how many bugs we detect if we test a particular fraction of the analysis groups.

outperforms `count` beyond the 10% in the ranking. It also reaches the 100% bug score earlier than the alternatives, although the difference is minimal.

The reason why DOWSER still requires 92% of the AGs to find *all* bugs, is that some of the bugs were very simple. The "simplest" cases include a trivial buffer overflow in `poppler` (worth 16 points), and two vulnerabilities in `sendmail` from 1999 (worth 20 points each). Since DOWSER is designed to prioritize complex array accesses, these buffer overflows end up in the low scoring group. (The "simple" analysis groups – with less than 26 points – start at 47.9%). Clearly, both heuristics provide much better results than random sampling. Except for the tail, they find the bugs significantly quicker, which proves their usefulness.

To summarize, we have shown that a testing strategy based on DOWSER scoring function is effective. It lets us find vulnerabilities quicker than random testing or a scoring function based on the length of an analysis group.

Symbolic execution

Table 2.2 presents attacks detected by DOWSER. The last section shows how long it takes before symbolic execution detects the bug. Since the vanilla version of S2E cannot handle these applications with the whole input marked as symbolic, we also run the experiments with minimal symbolic inputs ("*Magic* S2E"). It represents the best-case scenario when an all-knowing oracle tells the execution engine exactly which bytes it should make symbolic. Finally, we present DOWSER's execution times.

We run S2E for as short a time as possible, e.g., a single request/response in `nginx` and transcoding a single frame in `ffmpeg`. Still, in most applications, vanilla

S2E fails to find bugs in a reasonable amount of time. `inspircd` is an exception, but in this case we explicitly tested the vulnerable DNS resolver only. In the case of `libexif`, we can see no difference between "Magic S2E" and DOWSER, so DOWSER's guidance did not influence the results. The reason is that our test suite here was simple, and the execution paths reached the vulnerability condition quickly. In contrast, more complex applications process the inputs intensively, moving symbolic execution away from the code of interest. In all these cases, DOWSER finds bugs significantly faster. Even if we take the 15 minute tests of higher-ranking analysis groups into account, DOWSER provides a considerable improvement over existing systems.

2.7 Related work

DOWSER is a 'guided' fuzzer which draws on knowledge from multiple domains. In this section, we place our system in the context of existing approaches. We start with the scoring function and selection of code fragments. Next, we discuss traditional fuzzing. We then review previous work on dynamic taint analysis in fuzzing, and finally, discuss existing work on whitebox fuzzing and symbolic execution.

Software complexity metrics Many studies have shown that software complexity metrics are positively correlated with defect density or security vulnerabilities [95; 117; 48; 142; 117; 98]. However, Nagappan et al. [95] argued that no single set of metrics fits all projects, while Zimmermann et al. [142] emphasize a need for metrics that exploit the unique characteristics of vulnerabilities, e.g., buffer overflows or integer overruns. All these approaches consider the broad class of post-release defects or security vulnerabilities, and consider a very generic set of measurements, e.g., the number of basic blocks in a function's control flow graph, the number of global or local variables read or written, the maximum nesting level of `if` or `while` statements and so on. DOWSER is very different in this respect, and to the best of our knowledge, the first of its kind. We focus on a narrow group of security vulnerabilities, i.e., buffer overflows, so our scoring function is tailored to reflect the complexity of pointer manipulation instructions.

Traditional fuzzing Software fuzzing started in earnest in the 90s when Miller et al. [90] described how they fed random inputs to (UNIX) utilities, and managed to crash 25-33% of the target programs. More advanced fuzzers along the same lines, like Spike [123], and SNOOZE [15], deliberately generate malformed inputs, while later fuzzers that aim for deeper bugs are often based on the input grammar (e.g., Kaksonen [76] and [124]). DeMott [39] offers a survey of fuzz testing tools. As observed by Godefroid et al. [53], traditional fuzzers are useful, but typically find only shallow bugs.

Application of DTA to fuzzing BuzzFuzz [47] uses DTA to locate regions of seed input files that influence values used at library calls. They specifically select library

calls, as they are often developed by different people than the author of the calling program and often lack a perfect description of the API. Buzzfuzz does not use symbolic execution at all, but uses DTA only to ensure that they preserve the right input format. Unlike DOWSER, it ignores implicit flows completely, so it could never find bugs such as the one in nginx (Figure 2.1). In addition, DOWSER is more selective in the application of DTA. It's difficult to assess which library calls are important and require a closer inspection, while DOWSER explicitly selects complex code fragments.

TaintScope [132] is similar in that it also uses DTA to select fields of the input seed which influence security-sensitive points (e.g., system/library calls). In addition, TaintScope is capable of identifying and bypassing checksum checks. Like Buzzfuzz, it differs from DOWSER in that it ignores implicit flows and assumes only that library calls are the interesting points. Unlike BuzzFuzz, TaintScope operates at the binary level, rather than the source.

Symbolic-execution-based fuzzing Recently, there has been much interest in white-box fuzzing, symbolic execution, concolic execution, and constraint solving. Examples include EXE [20], KLEE [21], CUTE [114], DART [52], SAGE [53], and the work by Moser et al. [93]. Microsoft's SAGE, for instance, starts with a well-formed input and symbolically executes the program under test in attempt to sweep through all feasible execution paths of the program. While doing so, it checks security properties using AppVerifier. All of these systems substitute (some of the) program inputs with symbolic values, gather input constraints on a program trace, and generate new input that exercises different paths in the program. They are very powerful, and can analyze programs in detail, but it is difficult to make them scale (especially if you want to explore many loop-based array accesses). The problem is that the number of paths grows very quickly.

Zesti [88] takes a different approach and executes existing regression tests symbolically. Intuitively, it checks whether they can trigger a vulnerable condition by slightly modifying the test input. This technique scales better and is useful for finding bugs in paths in the neighborhood of existing test suites. It is not suitable for bugs that are far from these paths. As an example, a generic input which exercises the vulnerable loop in Figure 2.1 has the `uri` of the form "`//`*{arbitrary characters}*", and the shortest input triggering the bug is "`//../`". When fed with "`//abc`", [88] does not find the bug—because it was not designed for this scenario. Instead, it requires an input which is much closer to the vulnerability condition, e.g., "`//..`*{an arbitrary character}*". For DOWSER, the generic input is sufficient.

SmartFuzz [92] focuses on integer bugs. It uses symbolic execution to construct test cases that trigger arithmetic overflows, non-value-preserving width conversions, or dangerous signed/unsigned conversions. In contrast, DOWSER targets the more common (and harder to find) case of buffer overflows. Finally, Babić et al. [12] guide symbolic execution to potentially vulnerable program points detected with static analysis. However, the interprocedural context- and flow-sensitive static anal-

ysis proposed does not scale well to real world programs and the experimental results contain only short traces.

2.8 Conclusion

DOWSER is a guided fuzzer that combines static analysis, dynamic taint analysis, and symbolic execution to find buffer overflow vulnerabilities deep in a program's logic. It starts by determining 'interesting' array accesses, i.e., accesses that are most likely to harbor buffer overflows. It ranks these accesses in order of complexity—allowing security experts to focus on complex bugs, if so desired. Next, it uses taint analysis to determine which inputs influence these array accesses and fuzzes only these bytes. Specifically, it makes (only) these bytes symbolic in the subsequent symbolic execution. Where possible DOWSER's symbolic execution engine selects paths that are most likely to lead to overflows. Each three of the steps contain novel contributions in and of themselves (e.g., the ranking of array accesses, the implicit flow handling in taint analysis, and the symbolic execution based on pointer value coverage), but the overall contribution is a new, practical and complete fuzzing approach that scales to real applications and complex bugs that would be hard or impossible to find with existing techniques. Moreover, DOWSER proposes a novel 'spot-check' approach to finding buffer overflows in real software.

Acknowledgment

This work is supported by the European Research Council through project ERC-2010-StG 259108-ROSETTA, the EU FP7 SysSec Network of Excellence and by the Microsoft Research PhD Scholarship Programme through the project MRL 2011-049. The authors would like to thank Bartek Knapik for his help in designing the statistical evaluation.

Program	Vulnerability	Dowsing			Symbolic input	Symbolic execution		
		AG score	Loops	LoC		V-S2E	M-S2E	DOWSER
nginx 0.6.32	CVE-2009-2629 heap underflow	4th out of 62/140 630 points	517	66k	URI field 50 bytes	> 8 h	> 8 h	253 sec
ffmpeg 0.5	UNKNOWN heap overread	3rd out of 727/1419 2186 points	1286	300k	Huffman table 224 bytes	> 8 h	> 8 h	48 sec
inspircd 1.1.22	CVE-2012-1836 heap overflow	1st out of 66/176 625 points	1750	45k	DNS response 301 bytes	200 sec	200 sec	32 sec
poppler 0.15.0	UNKNOWN heap overread	39th out of 388/904 1075 points	1737	120k	JPEG image 1024 bytes	> 8 h	> 8 h	14 sec
poppler 0.15.0	CVE-2010-3704 heap overflow	59th out of 388/904 910 points	1737	120k	Embedded font 1024 bytes	> 8 h	> 8 h	762 sec
libexif 0.6.20	CVE-2012-2841 heap overflow	8th out of 15/31 501 points	121	10k	EXIF tag/length 1024 + 4 bytes	> 8 h	652 sec	652 sec
libexif 0.6.20	CVE-2012-2840 off-by-one error	15th out of 15/31 40 points	121	10k	EXIF tag/length 1024 + 4 bytes	> 8 h	347 sec	347 sec
libexif 0.6.20	CVE-2012-2813 heap overflow	15th out of 15/31 40 points	121	10k	EXIF tag/length 1024 + 4 bytes	> 8 h	277 sec	277 sec
snort 2.4.0	CVE-2005-3252 stack overflow	24th out of 60/174 246 points	616	75k	UDP packet 1100 bytes	> 8 h	> 8 h	617 sec

Table 2.2: Applications tested with DOWSER. The *Dowsing* section presents the results of DOWSER's ranking scheme. *AG score* is the complexity of the vulnerable analysis group - its position among other analysis groups; X/Y denotes all analysis groups that are "complex enough" to be potentially analyzed/all analysis groups which access arrays; and the number of points it scores. *Loops* counts outermost loops in the whole program, and *LoC* - the lines of code according to `sloccount`. *Symbolic input* specifies how many and which parts of the input were determined to be marked as symbolic by the first two components of DOWSER. The last section shows symbolic execution times until revealing the bug.

3

Scalable Data Structure Detection and Classification for C/C++ Binaries

Abstract

Many existing techniques for reversing data structures in C/C++ binaries are limited to low-level programming constructs, such as individual variables or `structs`. Unfortunately, without detailed information about a program's pointer structures, forensics and reverse engineering are exceedingly hard. To fill this gap, we propose MEMPICK, a tool that detects and classifies high-level data structures used in stripped binaries. By analyzing how links between memory objects evolve throughout the program execution, it distinguishes between many commonly used data structures, such as singly- or doubly-linked lists, many types of trees (e.g., AVL, red-black trees, B-trees), and graphs. We evaluate the technique on 10 real world applications, 4 file system implementations and 16 popular libraries. The results show that MEMPICK can identify the data structures with high accuracy.

3.1 Introduction

Modern software typically revolves around its data structures. Knowing the data structures significantly eases the reverse engineering efforts. Conversely, not knowing the data structures makes the already difficult task of understanding the program's code and data even harder. In addition, a deep knowledge of the program's data structures enables new kinds of binary optimization. For instance, an optimizer may keep the nodes of a tree on a small number of pages (to reduce page faults and TLB flushes). On a wilder note, some researchers propose that aggressive optimizers automatically replace the data structures themselves by more efficient variants (e.g., an unbalanced search tree by an AVL tree) [74], [75].

Accurate data structure detection is also useful for other analysis techniques. For instance, dynamic invariant detection [43] infers relationships between the values of variables. Knowing the types of pointer and value fields, for instance in a red-black tree, helps to select the relevant values to compare and to avoid unnecessary computation [59]. Likewise, principal components analysis (PCA) [69] is a technique to reduce a high-dimensional set of correlated data to a lower-dimensional set with less correlation that captures the essence of the full dataset but is much easier to process. PCA is used in a wide range of fields for many decades. In recent years, it has become particularly popular as a tool to summarize tree data structures [60; 11].

Unfortunately, most reversing techniques for data structures in C/C++ binaries focus on "simple" data types: primitive types (like int, float, and char) and their single block extensions (like arrays, strings, and structs) [59; 86; 119; 85]. They do not cater to trees, linked lists, and other pointer structures.

Existing work on the extraction of pointer structures is limited. For instance, the work by Cozzie et al. is unabashedly imprecise [34]. Specifically, they do not (and need not) care for precise type identification as they only use the data structures to test whether different malware samples are similar in their data structures. Of course, this also means that the approach is not suited for reverse engineering or precise analysis.

A very elegant system for pointer structure extraction, and perhaps the most powerful to date, is DDT by Jung and Clark [74]. It is accurate in detecting both the data structures and the functions that manipulate them. However, the approach is limited by its assumptions. Specifically, it assumes that the programs access the data structures exclusively through explicit calls to a set of access functions. This is a strict requirement and a strong limitation, as *inline* manipulation of data structures—without separate function calls—is common. Even if the programmer defined explicit access functions, most optimizing compilers inline short access functions in the more aggressive optimization levels. Also, in their paper, Jung and Clark do not address the problem of overlapping data structures—like a linked list that connects the nodes of a tree. Overlapping data structures, sometimes referred to as *overlays*, are very common also.

3.1.1 Contributions

In this paper, we describe MEMPICK: a set of techniques to detect and classify heap data structures used by a C/C++ binary. MEMPICK requires neither source code, nor debug symbols, and detects data structures reliably even if the program accesses them inline. Detection is based on the observation that the shape of a data structure reveals information about its type. For instance, if an interconnected collection of heap buffers *looks* like a balanced binary tree throughout the program's execution, it probably is one. Thus, instead of analyzing the instructions that modify a data structure, we observe dynamically how its shape evolves. As a result, MEMPICK does not make any assumptions about the structure of the binary it analyzes and handles binaries compiled with many different optimization levels, containing inline assembly, or using various function calling conventions.

Since our detection mechanism is based solely on the shape of data structures, we do not identify any features that characterize their contents. For instance, we cannot tell whether a binary tree is a binary search tree or not. Nor do we pinpoint the functions that perform the operations on data structures.

On the other hand, MEMPICK is suitable for all data structures that are distinguishable by their shape. The current implementation handles singly- and doubly-linked lists (cyclic or not), binary and n-ary trees, various types of balanced trees (e.g., red-black treed, AVL trees, and B-trees), and graphs. Additionally, we implemented measures to recognize sentinel nodes and threaded trees.

One of the qualitatively distinct features of MEMPICK is its generic method for dealing with overlays (overlapping data structures). Overlays complicate the classification, as the additional pointers blur the shape of the data structures. However, even if all nodes in a tree are also connected via a linked list, say, MEMPICK will notice the overall data structures and present them as a "tree with linked list" to the user.

Since MEMPICK relies on dynamic analysis, it has the same limitation as all other such techniques—we can only detect what we execute. In other words, we are limited by the code coverage of the profiling runs. While code coverage for binaries is an active field of research [28; 53], it is not the topic of this paper. In principle, MEMPICK works with any binary code coverage tool available, but for simplicity, we limited ourselves to existing test suites in our evaluation.

3.1.2 Outline

We start by giving more context about binary analysis in Section 3.2 and provide two example applications in Section 3.3. In Section 3.4 we describe the overall architecture of the system. Section 3.5 presents details about the low-level manipulation of the memory graph, that is the basis of our high level data structure representation from Section 3.6. Section 3.7 deals with the intricacies of data structure classification, that are extended with additional details about height balanced trees

in Section 3.8. In Section 3.9 we give an overview of information offered to the user, followed by an extensive evaluation in Section 6.9. We also discuss the computational complexity in Section 3.11 to argue about the scalability of the proposed approach. In Section 4.2.2 we look at the observed limitations and possible extensions to MEMPICK. Finally we discuss related projects on data structure reverse engineering in Section 3.13 and conclude the paper in Section 5.5.

This paper is an extended version of our WCRE 2013 publication [61].

3.2 Static versus dynamic analysis

There are two main approaches to reverse engineering, static and dynamic analysis. In this section we discuss the pros and cons of each for data structure discovery and explain why we opted for dynamic analysis.

Static analysis reasons about the application without executing it. This enables the analysis of components that are difficult to execute normally. While static analysis is the most suitable to reason about the application as a whole, it is fundamentally imprecise for weakly typed languages such as C/C++/Assembly due to pointer aliasing and indirect control flow changes. This typically manifests itself in either false positives [73] or false negatives [42] depending on the analysis model. In the area of binary analysis even the most powerful static technique has problems handling even the most basic aggregate arrays [14]. On the other hand dynamic analysis is inherently context sensitive since it reasons about a concrete execution path. While this approach can only cover what is executed, the added run-time information improves the precision and the scalability of MEMPICK. In the following we discuss the validity of detecting pointer structures using only dynamic analysis.

Complex data structures represent the core of most algorithms, and are thus used pervasively throughout the application. Data structures implementations are also typically designed as reusable software components, and employed in multiple contexts within the same application. In consequence, complete code coverage is not an appropriate measure for the effectiveness of data structure discovery. Functional coverage, which measures coverage in program features is more relevant for this type of analysis. We believe that in practice a reasonable sized set of unit tests can provide enough functional coverage to extract the core data structures using MEMPICK. Our evaluation confirms this assumption, since we discovered more than 100 different data structures in 10 applications using basic inputs.

3.3 Example applications

To demonstrate the usefulness of MEMPICK, we describe two possible applications: malware analysis and retrofitting security to legacy binaries not designed with security in mind. We target MEMPICK at binary analysis where no source code or high level information is available directly from the application. This scenario is typical

of malware analysis, which is critical in taking down large scale malicious infrastructures such as botnets [111]. MEMPICK is also applicable to benign applications, not for reverse engineering their contents, but to discover and secure existing structures [120].

3.3.1 Case study 1: Malware analysis

With the transition towards online services, there is an ever greater incentive to infect user machines with different varieties of malware. Botnets go beyond the traditional single instance attacks, connecting instead the infected machines in a custom network, used to control and monetize the system. Security experts and law enforcement on the other hand aim to infiltrate and disrupt this network in a botnet takedown. This process is exceedingly difficult due to the encryption and custom protocols employed in botnets.

Rossow et. al. [111] provide an insightful analysis on the resilience of the state-of-the art Peer-To-Peer botnet families and potential avenues of attack. This work relies heavily on the reverse engineering of the communication protocols and the underlying encryption algorithms. This process is currently mostly manual effort relying on the experience of the reverse engineer to discover high level code structures. With MEMPICK, we aim to provide additional information about pointer structures in order, to complement the low-level data structure information from Howard [119]. Pointer structures are highly relevant, since malware designers can easily integrate them to distribute information within different memory objects. Further extensions of MEMPICK could also detect and semantically annotate the code manipulating the data structure, allowing the reverse engineer to focus his attention on application logic instead.

3.3.2 Case study 2: Security hardening for third party applications

System administrators frequently have to deal with the integration of closed-source components into their systems, such as third party libraries or device drivers. These components may contain vulnerabilities, that impact the security of the whole system. For example researchers at Microsoft confirm that many vulnerabilities in device drivers stem from the misuse of pointer structures [136]. Recent research in system security started to shift focus from compiler based security enhancements towards binary hardening to overcome this challenge [120; 141].

MEMPICK detects the pointer data structures within the potentially vulnerable code as a starting point for the hardening process. The vulnerabilities include race conditions on the data structure, as well as malicious out-of-band changes by an attacker. Slowinska et. al. [120] developed a solution that leverages low-level data structure detection to harden these constructs against potential attacks. In the future we hope to infer semantic information about the underlying interface functions as-well. This will enable fully automatic monitoring of data structure validity to

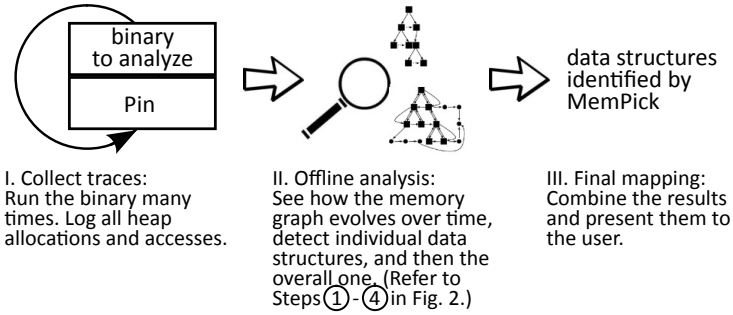


Figure 3.1: MEMPICK: high-level overview.

uncover attacks or bugs at run-time.

3.4 MEMPICK

We now discuss our approach in detail. Throughout the paper, we will use the data structure in Figure 3.2 as our running example. The example is a snapshot of a binary tree with three overlapping data structures: a child tree, a parent tree and a list facilitating tree traversal. Each node of the tree has a pointer to a singly-linked list of some unrelated objects that ends with a sentinel node. The example is sufficiently complex to highlight some of the difficulties MEMPICK must overcome and sufficiently simple to track manually.

Figure 3.1 illustrates a high-level overview of MEMPICK. The detection procedure consists of three major stages. First, we record sample executions of an application binary. Next, we feed each of them to an offline analysis engine that identifies and classifies the data structures. Finally we combine the results and present them to the user.

The first stage requires tracking and recording the execution of the application, and for this we use Intel's PIN binary instrumentation framework [71]. PIN provides a rich API that allows monitoring context information, e.g., register or memory contents, for select program instructions, function- and system calls. We instrumented Pin to record memory allocation functions, along with instructions storing the addresses of all buffers allocated on the heap. In the remainder of this paper, we assume that applications use general-purpose memory allocators like `malloc()` and `free()`, or `mmap()`. It is straightforward to use the approach by Chen et al. [26] to detect custom memory allocators and instrument those instead.

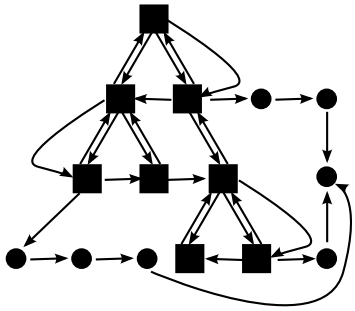
For the offline analysis stage, MEMPICK analyzes the shape of links between heap buffers to identify the data structures. It consists of four further steps. In this section, we describe them briefly and defer the details to later sections (see also the circled numbers in Figure 3.2).

① MEMPICK first organizes all heap buffers allocated by the application, along

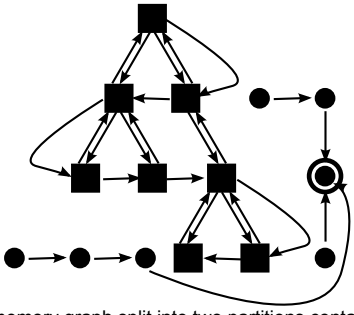
```
typedef struct list_node {
  data_t data;
  lnode_t *next;
} lnode_t;

typedef struct tree_node {
  data_t data;
  tnode_t *left, *right;
  tnode_t *parent, *next;
  lnode_t *list_elem;
} tnode_t;
```

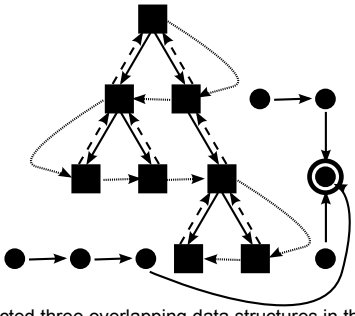
Data structures used in the memory graph. As MemBrush operates at the binary level, it does not have access to this information.



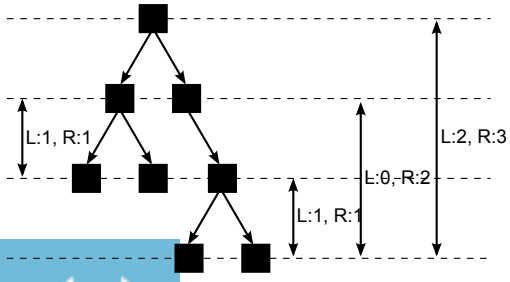
(1) (a) The memory graph after MemPick identified the types of the objects.



(2) (b) The memory graph split into two partitions containing objects of the same type. MemPick also found one sentinel node (denoted by the double circle).



(3) (c) MemPick detected three overlapping data structures in the collection of the squared nodes: a child tree, a parent tree, and a list. Each of them is denoted by a different line type. Next, the collection of the squared nodes is classified as a binary tree with a linked list, and the collection of the circled nodes as three sinlgly-linked lists.



(4) (d) MemPick measures if the tree is balanced. It measures the height of the subtrees recursively, and concludes that it is unbalanced.

Figure 3.2: A running example illustrating MEMPICK's detection algorithm.

with all links between them, into a *memory graph*. It reflects how the connections between the buffers evolve during the execution.

② Next, MEMPICK performs type analysis to split the graph into collections of objects of the same type. For instance, Figure (3.2a) illustrates a fragment of the memory graph at a point when the tree contains 8 nodes, and Figure 3.2b partitions the data structure into objects of the same type.

③ Given the partitions, MEMPICK analyzes the shape of the data structures by considering the links in each partition, searching for overlapping structures, and finally identifying types. For example, in Figure (3.2c), all squared nodes would end up in one partition. Since it is common for data structure implementations to use auxiliary pointers, e.g., to form a parent tree, or a list facilitating traversal, the overall shape can become convoluted, such as it is the case in Figure (3.2c). By looking at the overlapping structures, MEMPICK first learns that the collection of squared nodes contains a child tree, a parent tree, and a list, while each circled partition is a list. It then classifies the first data structure as a binary tree by means of a decision tree.

④ Finally, MEMPICK measures how each tree used by the application is balanced in order to distinguish between various types of height-balanced trees, e.g., red-black and AVL trees. This is illustrated in Figure (3.2d).

We discuss each step in detail in Sections 3.5-3.8. Once all the execution traces are analyzed, we combine the results, and present them to the user (Section 3.9).

3.5 Memory Graphs: Interconnected Heap Objects

A memory graph illustrates how links between heap objects evolve during the execution of an application. By itself, this is not enough to extract the links that are relevant to identify a data structure. For instance, in Figure 3.2a, we do not want to report a graph comprising all the nodes, but rather classify the tree and the lists separately. For this purpose, MEMPICK strips connections between nodes featuring different logical types. We introduce the term logical type, to be able to reason about a weak form of type equality, necessary when dealing with potentially polymorphic types. Two objects share a logical type if either their low-level C/C++ type is identical or one object can be used in place of another, i.e. `sub-class`es or equivalent behavior in C.

Building the graph Like RDS [109] and DDT [74], MEMPICK inserts new nodes in the graph whenever a heap allocation occurs, and deletes existing ones upon deallocation. Edges represent connections between the allocated buffers. MEMPICK adds or removes them whenever the application modifies a link between two heap objects. This happens either on instructions that store a pointer to one object in another one, on instructions that clear previous pointers, or on calls to the memory deallocation functions.

Tagging the graph with type information Conceptually MEMPICK assigns two objects the same logical type if they could both be used in the same operand position of a given instruction. This follows the intuition that an instruction carries implicit typing of its operands. However, to avoid false-positives (classifying two different objects together), MEMPICK first excludes instructions that might be *type agnostic* and handle objects of various types, such as instructions in `memcpy`-like functions.

Next we describe the type inference algorithm in more detail. The algorithm initially associates each instruction with a pair of unique tags, one for its source and one for its destination operand. Whenever a heap object is used as operand for a given instruction, it inherits the corresponding instruction tag. This ensures that different heap objects used within the same instruction operand all share a unique tag. If the heap object is already associated with another tag, than both tags correspond to the same logical type, and have to be merged into one. The tag change is then propagated across all heap objects to ensure consistency. While this algorithm is very simple, it successfully captures our assumptions about logical types. Even though the system starts out with zero assumptions and a highly fragmented tag space, the merging operation quickly converges and identifies a small set of logical types within the program.

During this algorithm it is critical to avoid any instructions with ambiguous typing. To do so, MEMPICK classifies an instruction as *type aware* if it consistently stores a pointer to a heap buffer (or `NULL`) to a memory location at a specific, constant offset in another heap buffer. In other words, we do not consider instructions that store non-pointer values to the heap objects, or that store the pointers at different offsets at different times, etc. However, as it is common for applications to keep sentinel nodes in static or stack memory, we need to relax these filtering condition a little to allow for pointers to sentinel nodes.

The way MEMPICK extends the memory graph with type information is different from the approach used by DDT [74]. In particular, DDT applies typing based on allocation site, which poses problems when an application allocates an object of the same type in multiple places in the code—which is quite common in real software. For example, linked lists allocate memory nodes when inserting elements both in the STL (`push_front` and `push_back`) and the GNOME GLib (`g_list_append` and `g_list_prepend`) libraries. To handle these cases, DDT further examines if objects from different allocation sites are modified by the same interface functions in another portion of the application. As discussed in Section 3.1, relying on the interface is a strong assumption that fails in the case of code that uses macros, say, rather than function calls to access the data structures, or in the face of aggressive optimization where the access code is inlined.

If necessary, we can further refine our analysis with existing approaches to data structure detection, like Howard [119] or static analyses [13; 108; 110]. While none of these techniques can combine objects from different allocation sites, they would help reduce possible false positives. For all experiments in this paper, however, we

use MemPick's mechanism exclusively.

3.6 From Memory Graph to Individual Data Structures

Given the memory graph, MEMPICK divides it into subgraphs, each containing individual data structures. These data structures will form the basis for our shape analysis (see Section 3.7). As illustrated in Figure (3.2b), the output partitions are connected subgraphs whose nodes all have the same type.

MEMPICK starts by removing from the graph all links that connect nodes of different types. Doing so splits the graph into components that each reflect transformations of individual structures during execution. In the example from Figure 3.2, one of the partitions would illustrate the growth of the tree.

Observe that not every snapshot of the memory graph is suitable for shape analysis. The problem is that properties characteristic to a data structure are not necessarily maintained at each and every point of the execution. For example, if an application uses a red-black tree, the tree is often *not* balanced just before a rotate operation. However, in all the *quiescent* time periods when the application does not modify the tree, its shape retains the expected features, e.g., it is balanced, every node has at most one parent, there are no cycles, and so on. Therefore, MEMPICK performs its shape analysis only when a data structure is quiescent.

MEMPICK defines quiescent periods in the number of instructions executed. Specifically, we measure the duration (in cycles) of the gaps between modifications of the data structures and then pick the longest n% as the quiescent periods. As long as we are sufficiently selective, we will never pick non quiescent periods. For instance, in our experiments, we picked only the longest 1% gaps as quiescent periods. The dynamic gap size allows MEMPICK to adapt to the characteristics of each binary and data structure. Compiler options and data usage patterns all contribute to the observed gap sizes. The method defined in MEMPICK benefits from two core properties, 1) it guarantees a lower bound of quiescent periods for every data structure, 2) it provides maximum robustness, by selecting the largest possible gap size that still satisfies the quiescent period frequency desired by the reverse engineer.

Before we pass the stable snapshots of each data structure to the shape analyzer, we detect and disconnect sentinel nodes. The problem of sentinels is that they blur the shape of data structures. For example, if we did not disconnect the sentinel node in Figure (3.2b), it would be difficult to see that the partition of circled nodes is in fact a collection of three lists.

To pinpoint sentinel nodes in a partition of the memory graph, MEMPICK counts the number of incoming edges for each node, and searches for outliers. While this strategy works well for lists, trees, and graphs, it might break some highly customized data structures. For example, in the case of a star-shaped data structure, it disconnects the central node, and MEMPICK reports a collection of lists.

Finally, for each partition of the memory graph, we acquire its snapshots in the

quiescent periods, and use them in the following stage of MEMPICK's algorithm, discussed in the next section.

3.7 Shape Detection

We identify the shape of the graph-partitions based on observations during the quiescent periods. As we described above, MEMPICK focuses on quiescent periods since they represent the stable state of the data structure. Any given shape hypothesis needs to hold for every "snapshot" of the graph-partition, since it represents a globally valid property of the data structure. Outliers are not allowed, as they would reduce the certainty of the final hypothesis. Since data structures often overlap and each of the overlapping substructures blur the actual shape, we identify them first. For instance, in Figure (3.2c), it is not simple to tell that the component composed of squares actually represents a tree. Only after we distinguish between the child tree, the parent tree, and so on, does the identification become straightforward. Given the overlapping structures, we employ a hand-crafted decision tree that finally classifies the data structure. As a final step, we offer support for refined classifiers cases that discover data structures requiring more advanced analysis, e.g., threaded trees. We now discuss the stages in turn.

3.7.1 Overlapping Data Structure Identification

To find overlapping data structures, we search for sets of pointer variables that keep all nodes of the data structure connected. To exclude unnecessary pointers from such sets, we define the term `minimal pointer set`. Each of these sets features the following properties: *the subgraph generating by maintaining only the edges corresponding to the pointers remains connected; also no subset of a minimal pointer set holds the first property*. Intuitively eliminating any entry from such a set leads to a disconnected data structure. The problem of finding the minimal pointer set is analogous to the maximal set and set cover problems in complexity theory. In the remainder of this section, we refer to the constituent overlapping data structures as *overlays*, following a similar notion in network graphs.

In particular, for each partition of the memory graph, we consider a set $\mathcal{P} = \{p_1, \ldots, p_n\}$, where each p_i is the offset of a pointer variable in the `struct` or `class` representing the node type. For example, in Figure 3.2, we have $\mathcal{P} = \{4, 8, 12, 16\}$, which maps to the set of pointers in the tree: {`left`, `right`, `parent`, `next`}. Next, we list all maximal subsets of $\mathcal{P}$ that keep the partition connected. The subsets are maximal in the sense that they do not contain any redundant elements, i.e., if we remove an element from a subset, the remaining pointers do not cover the whole partition. In the tree in Figure 3.2, we identify the following set of overlays: $\{\{4, 8\}, \{12\}, \{16\}\}$. The first overlay uses the `left` and `right` pointers to connect the tree from a top-down perspective. The second overlay uses the `parent` pointer

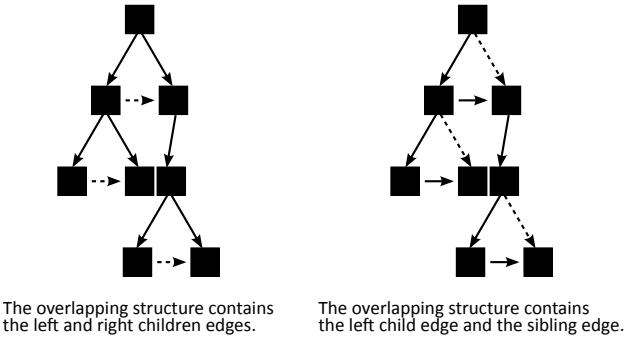


Figure 3.3: An example binary tree with three pointer variables: `left`, `right`, and `sibling`. It has three overlapping data structures: {`left`, `right`} depicted on the left (denoted by the solid edges), {`left`, `sibling`} depicted on the right and {`right`, `sibling`}. Observe that the overlapping data structures are not disjoint — both contain the left child edge.

for the potential of a bottom-up traversal. The last overlay is an alternate linked-list style overlay using `next` pointer to access all nodes without having to traverse the tree. While the overlays in this example involve a disjoint set of pointers, in practice pointers may also be shared, as it is the case in the example from Figure 3.3. In this example neither pointer can form an overlay by itself, but any combination of two results in a connected data structure and should be reported for further analysis.

Finally, overlays are also interesting for non-tree shapes, as the can help to disambiguate non-standard list variants. For example, the non-circular doubly-linked list within the `utlist` library uses a peculiar implementation. While the `next` pointer is implemented as a traditional non-circular linked list, the `prev` pointer is made to be circular. Such hybrid implementations can difficult to identify, due to the lack of symmetry, without separating the individual overlays.

Once the overlays are identified, we apply the rules in Table 3.1 to classify each overlay individually. Columns 2–5 specify the number of incoming and outgoing edges for ordinary and special nodes, while the last one defines how many special nodes there are . For instance, each “ordinary” (internal) node of a list has one incoming and one outgoing edge; additionally each list has one node with just one outgoing edge (the head), and one node with just one incoming edge (the tail). Currently, we do not distinguish between different classes of graph, e.g., cyclic and acyclic graphs, but extending the list of rules is straightforward.

3.7.2 Data Structure Classification

Finally, MEMPICK combines the information about all overlays, and reports a high-level description of the partition being analyzed. This step follows a decision tree presented in Figure 3.4. To classify the tree in Figure (3.2c), MEMPICK first checks

Table 3.1: MEMPICK's rules to classify individual overlays. They specify the number of incoming and outgoing edges for ordinary and special nodes.

Type	Ordinary nodes		Special nodes		
	in	out	in	out	#
List	1	1	0	1	1
			1	0	1
Circular list	1	1	–	–	–
Binary child tree	1	{0,1,2}	0	{1,2}	1
Binary parent tree	{0,1,2}	1	{1,2}	0	1
3-ary child tree	1	{0,...,3}	0	{1,...,3}	1
3-ary parent tree	{0,...,3}	1	{1,...,3}	0	1
n-ary child tree	1	{0,...,n}	0	{1,...,n}	1
n-ary parent tree	{0,...,n}	1	{1,...,n}	0	1
Graph	all the remaining cases				

that the data structure has no graph overlays. Since it contains a binary child and a parent tree overlays, MEMPICK reports a binary tree (with an additional linear overlay). To refine the results, MEMPICK additionally measures the balance of the tree in Section 3.8.

3.7.3 Refinement Classifiers for Special Data Structures

Some popular data structures have very specific shapes for which the general classification rules of Section 3.7.2 are not sufficient. Threaded trees are one such example, currently support by MEMPICK. In order to increase the accuracy of it classification, MEMPICK allows for the addition of refinement classifiers that are tailored for specific data structures. We will discuss the threaded tree as an example, as it is the only common data structure with an "exceptional" shape we encountered throughout our extensive evaluation (Section 3.10).

Threaded trees represent a variation on the binary tree representation that is used in practice. In a threaded tree, all child pointers that would be null in a binary tree, now point to the in-order predecessor or the in-order successor of the node. For instance, the left child could point to the predecessor and the right child to the successor. Alternatively, the data structure may use only one of the children for threading. Without loss of generality, assume the threaded tree uses the right child node to thread to the successor node. Threading facilitates tree traversal, without relying on parent pointers or recursion. The additional links are known as *threads*, and can use either one or both child pointers. Refer to Figure 3.5 for an example threaded tree. In our experiments, threaded trees appear in three libraries, including the GNOME GLib library.

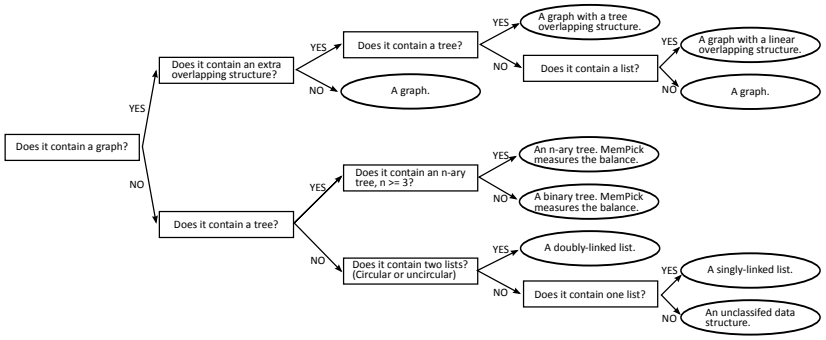


Figure 3.4: MEMPICK's decision tree used to perform the final classification of a partition of the memory graph.

Since a threaded child pointer keeps all nodes of the tree connected, it forms a single element overlapping structure. Observe that it has the shape of a binary parent tree. Thus, a child pointer is either threaded, and it forms a binary parent tree overlapping structure itself, or it is not threaded, and never included in an overlapping structure. Since it is an extraordinary situation, MEMPICK performs further analysis to test if this partition of the memory graph is a threaded tree. It searches for a root candidate to which it can successfully apply the *un-threading* algorithm [135]. After this step, it obtains a binary child tree overlapping structure, and the final classification is straightforward.

3.8 Classification of Height-balanced Trees

Once the shape detection step in Section 3.7.2 classifies a data structure as a tree, MEMPICK also attempts to reason about the properties of the tree to allow for a more precise classification. We have identified that the size and height properties of the different sub-trees within a tree offer enough information for more detailed classification of height-balanced trees [31]. These properties also map well to the shape-only classification algorithm presented so far. MEMPICK recognizes height-balanced trees [31], and identifies AVL, red-black and B-trees. AVL trees are binary trees where the heights of the child subtrees of any node differ by at most one. Red-black trees on the other hand allow the height of the longest branch to be at most two times that of the shortest branch. B-trees combine high child count with perfect balance, all leaf nodes are located at the same height. Other balanced tree variants, like binomial or 2-3-4 trees can also be described based on their worst case imbalance and maximum child count.

MEMPICK measures the height of a tree recursively, starting at the root (i.e., the node with no incoming edges). While computing the height of the left and right subtrees, h_L and h_R, respectively, MEMPICK keeps track of both the absolute and

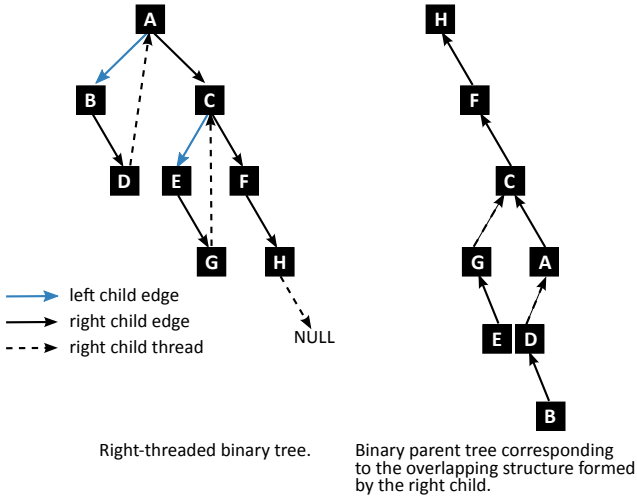


Figure 3.5: The left-hand side figure presents an example right-threaded binary tree, and the right-hand side one illustrates the corresponding overlapping binary parent tree.

relative height imbalance, i.e., $|h_L - h_R|$ and h_L/h_R. It classifies the tree as an AVL tree, if for all its subtrees $|h_L - h_R| \leq 1$, and as a red-black tree — under the condition that $\frac{1}{2} \leq h_L/h_R \leq 2$. For non-binary trees it checks the B-tree property of $|h_L - h_R| = 0$. Since MEMPICK focuses its analysis on the shape of a tree, it might misclassify a *too* balanced red-black tree as an AVL tree. However, if a red-black tree is always perfectly balanced, this behavior is very useful for an analyst to know about.

3.9 Final Mapping

In the final step, MEMPICK combines the results of the partitions from the different execution traces, and presents them to the user. The summaries generated by MEMPICK include each unique partition classification and their occurrence count. Thus no outliers are excluded, while the user is not overwhelmed with 100s of copies of the same classification.

MEMPICK can also project the detected data structures to local or global variables of the application. Whenever the application stores a pointer to an identified heap buffer in either stack or static memory, MEMPICK maps the destination to the stack frame of the currently executing function or a global memory location, respectively.

Since MEMPICK already operates with the notion of multiple independent partitions for a given data structure, it is straightforward to merge information from multiple runs. The only requirement for this process is the ability to detect types globally, between different application runs. The type inference engine in MEM-

PICK supports this requirement, by operating at the level individual instructions, which do not change between runs. The resulting global types allow MEMPICK to merge the results from multiple runs transparently to the user.

In the future, we plan to merge MEMPICK with Howard [119], a solution to extract low-level data structures from C binaries. As Howard automatically generates new debug symbol tables for stripped binaries, MEMPICK's results will fit in perfectly, providing the user with more detailed information about the data structures used by the application. For instance, in Figure 3.2, instead of information that the application has a pointer to a `struct` consisting of one integer and five other pointers, the user will learn that the application has a pointer to an unbalanced binary tree with three overlays, and that each node of the tree has a pointer to a singly-linked list of some other data structures. However, this extension is beyond the scope of this paper.

3.10 Evaluation

We evaluated MEMPICK on two sets of applications. For the first set, we gathered as many popular libraries for lists and trees as we could find. We then exercised the most interesting/relevant functions using test programs from the libraries' test suites. These synthetic tests allow us to control exactly the features we want to test. We also evaluated the quiescent period detection mechanism on these libraries to identify the requirements for gap size selection. Next, we evaluated MEMPICK on a set of real-world applications, like `chromium`, `lighttpd`, `wireshark`, and the `clang compiler`. We also evaluate the analysis time for these applications to show the scalability of the proposed approach. Finally we also looked into low-level system code with two major file system implementations, `ZFS` and `ntfs`.

3.10.1 Popular Libraries

We tested MEMPICK on 16 popular libraries that feature a diverse set of implementations for a wide range of data structures. Including libraries in the evaluation has multiple benefits. Firstly, they provide strong ground-truth guarantees since the data structures and their features are well documented. Secondly they provide a means to evaluate a wider range of implementation variants, since most applications typically rely on a few standard implementations like STL and GLib in practice. For all the libraries we tried to use the built-in self-test functionality. Only if such a test was not available, we built simple test harnesses to probe all functionalities.

In the following we present a short summary of the reasoning behind some of the library choices. The evaluation set contains 4 major STL variants and GLib, the libraries typically used by major Linux applications. In addition, libavl brings a large variety of both balanced and unbalanced binary trees, with different overlay configurations, like the presence of parent pointers or threadedness. Several libraries

(like UTlist, BSD queues, and Linux lists) implement inline data structures with no explicit interface in the final binary—typically by offering access macros instead of functions. We also include the Google implementation of in-memory B-Trees to validate the ability of MEMPICK to detect balanced non-binary trees. Typically B-Trees are implemented in database applications, which operate on persistent storage, leading to a lack of pointers in the data structure nodes.

Table 3.2 presents a summary of our results gathered from the libraries. We do not present results for individual data structure partitions as that number is dependent on the specific test applications. In all scenarios MEMPICK classified all partitions for any given data structure the same way.

For all tests executed, we encountered a total of two misclassifications, while all other data structures were successfully identified by MEMPICK (no false negatives). In the case of GDSL, the shape of the misclassified binary tree is detected appropriately, however MEMPICK reports perfect balancedness since the tree is limited to 3 nodes. The results is still valuable, since MEMPICK reports all other classification details accurately, this error is also unlikely to occur in applications that deal with real data. The misclassification in GLib is more subtle. The implementation of the N-ary tree uses parent, left child and sibling pointers. For optimization purposes the authors also include a *previous* pointer in the sibling list. MEMPICK correctly identifies the presence of an N-ary parent pointer and binary child pointer (left child + next sibling) trees, but it also detects an overlay using the left child and previous sibling pointers. This overlay does not match any basic shape and is reported as a graph, bringing the overall classification to a graph. Since MEMPICK also reports the overlay classification to the user, a human reverse engineer can accurately interpret the results. Alternatively, the user can add a (trivial) refinement classifier for this scenario, since the presence of two overlays does imply more structure than a generic graph, but we wanted to keep the number of refinement classifiers to a minimum. One observation is that both errors were found in libraries supporting a large variety of data structure implementations. This is not surprising, since the chance of non-standard data structures is increased with the size of the library. Still our results show that the overlay based classifier is resilient to unexpected data structure shapes, by correctly classifying all basic overlays contained within. Even if the overall classification fails, the partial results are still beneficial as an anchor for the reverse engineering process.

When testing the `utlist` library, we ran across an interesting classification report. MEMPICK reported a cyclic and a non-cyclic list overlay for the non-cyclic doubly linked list in the library. This behavior was confirmed to be a design decision, when correlating the results with the source code. This example shows the importance of the overlay based classification employed in MEMPICK. Without this approach the observed shape and behavior would not match any assumption about linked lists as the overall structure is neither properly cyclic nor non-cyclic.

Table 3.2: MEMPICK's evaluation across 16 libraries. #Total is the number of implementation variants of the given type available in the library, #TruePos is the number of correctly classified variants, #FalsePos is the number of misclassified variants

Library	Type	#Total	#TruePos	#FalsePos
boost:container	dlist	1	1	0
	RB tree	1	1	0
clibutils	slist	1	1	0
	dlist	1	1	0
	RB tree	1	1	0
GDSL	dlist	2	2	0
	binary tree	3	2	1
	RB tree	1	1	0
GLib	slist	1	1	0
	dlist	1	1	0
	binary tree	1	1	0
	AVL tree	1	1	0
	n-ary tree	1	0	1
gnulib	dlist	1	1	0
	RB tree	2	2	0
	AVL tree	2	2	0
google-btree	B-tree	1	1	0
libavl	binary tree	4	4	0
	RB tree	4	4	0
	AVL tree	4	4	0
LibDS	dlist	1	1	0
	AVL tree	1	1	0
linux/list.h	slist	1	1	0
	dlist	2	2	0
linux/rbtree.h	RB tree	1	1	0
queue.h	slist	2	2	0
	dlist	2	2	0
SGLIB	slist	1	1	0
	dlist	1	1	0
STDCXX	dlist	1	1	0
	RB tree	1	1	0
STL	dlist	1	1	0
	RB tree	1	1	0
STLport	dlist	1	1	0
	RB tree	1	1	0
UTlist	slist	1	1	0
	dlist	2	2	0

Summarizing these results, we see that MEMPICK successfully deals with a large variety of data structure implementations. It is capable of correctly identifying the underlying type, independent of the presence of interface functions and independent of overlay variations. The results also show the efficiency of classifying balanced

binary trees based only on shape information, provided the tree is sufficiently large.

Table 3.3: MEMPICK's gap size evaluation across 16 libraries. The percentages represent the gap size percentile used for quiescent period selection. The columns represent the number of data structure implementations affected, compared to the base-line of using 1% gaps in table 3.2

Library	Type	5%	10%	15%	20%
boost:container	dlist	0	0	0	0
	RB tree	0	0	0	1
clibutils	slist	0	0	0	0
	dlist	0	0	0	0
	RB tree	0	0	1	1
GDSL	dlist	0	0	0	0
	binary tree	1	1	1	1
	RB tree	0	1	1	1
GLib	slist	0	0	0	0
	dlist	1	1	1	1
	binary tree	0	1	1	1
	AVL tree	0	1	1	1
	n-ary tree	0	1	1	1
gnulib	dlist	0	0	1	0
	RB tree	0	1	2	0
	AVL tree	0	1	2	0
google-btree	B-tree	0	1	1	1
libavl	binary tree	2	2	2	2
	RB tree	2	2	2	2
	AVL tree	2	2	2	2
LibDS	dlist	0	0	0	0
	AVL tree	1	1	1	1
linux/list.h	slist	0	0	0	0
	dlist	0	0	0	0
linux/rbtree.h	RB tree	0	0	1	1
queue.h	slist	0	1	2	2
	dlist	2	2	2	2
SGLIB	slist	0	1	1	1
	dlist	0	1	1	1
STDCXX	dlist	0	0	0	0
	RB tree	0	0	0	1
STL	dlist	0	0	0	0
	RB tree	0	0	1	1
STLport	dlist	0	0	0	0
	RB tree	0	0	1	1
UTlist	slist	0	0	1	1
	dlist	0	0	0	0

Next we aimed to identify the impact of the gap size percentile used with quiescent periods. In Section 3.6 we suggested the use of 1% longest gaps as signal for quiescent periods. In this part of the evaluation we vary this number all the way to 20% and observe its impact on classification accuracy. We perform this part of the evaluation on libraries, instead of applications, since they offer more precise ground truth information. Table 3.3 presents the results, counting for all the data structure implementations where the classification was degraded in comparison to the original proposal. In some instances this degradation was in the form of missing overlays, while in other instances MEMPICK was unable to offer any valid classification. In general, one can observe that search-trees are more sensitive to the gap size, especially the implementations within the libavl library. This library offers a cloning interface for trees, which in some implementation variants does not respect the validity of the tree throughout the operation. If a quiescent period intervenes during the clone operation, the system will observe an invalid tree. One must also take into account, that we tested the libraries using the built-in unit tests whenever they were available. In this scenario data structure operations are typically executed in quick sequence, without any intervening application code. This explains the gap size sensitivity for some of the list implementations. Overall, these results suggest that the quiescent periods should be considered conservatively, especially in the presence of heavily used, complex data structures. Our suggestion when performing manually assisted reverse engineering is to start with a highly conservative gap size, which can progressively be increased to detect data structures potentially missed by the initial setting.

3.10.2 Applications

MEMPICK is designed as a powerful reverse-engineering tool for binary applications, so it is natural to evaluate its capabilities on a number of frequently used real applications. For this purpose we have selected 10 applications from a wide range of classes, including a compiler (Clang), a web browser (Chromium), a webserver (Lighttpd), multiple networking and graphics applications. Table 3.4 presents the number of code lines for each of these applications, giving an idea of their size.

As we discussed in the section 3.4, MEMPICK operates under the assumption that it can track all memory allocations. Two of the selected applications, namely Clang and Chromium, use custom memory allocators to manage the heap. In the case of Clang we also instrumented the custom memory allocators to gain insight to the internal data structures. For Chromium we were currently unable to perform such instrumentation. MEMPICK was still able to detect a large number of data structures that are defined in third-party libraries which still employ the system allocation routines. In principle, it would be straightforward to detect custom memory allocators automatically using techniques developed by Chen et al. [26].

Table 3.4: Number of C/C++ lines of code for the 10 real-world applications, excluding potential third party libraries.

Application	Version	Lines of code
chromium	29.0.1548	4190k
clang	3.2	1045k
inkscape	0.48.4	396k
lighttpd	1.4.32	40k
pachi	10.0	13k
povray	3.7.0.RC7	106k
quagga	0.99.22	194k
tor	0.2.4.12-alpha	119k
wget	1.14	68k
wireshark	1.10.0	1727k

Table 3.5 presents an overview of the results from all applications. It is important to note that for applications there exists no ground-truth information that we can compare against. For every application reported by MEMPICK we manually checked the corresponding source code to confirm the classification. We report two types of errors in table 3.5. One is typing errors, when a given data structure is misclassified by MEMPICK. The other is partition errors. They refer to data structures that were classified accurately overall, but for which a number of their partitions contained errors.

The accuracy of MEMPICK is demonstrated by the fact that only 3 type misclassifications were detected in all tests on all 10 applications. MEMPICK was successful in identifying a wide-range of data structures, from custom designed singly-linked lists to large n-ary trees used for ray-tracing. MEMPICK also highlights different developer trends in the use of data structures. Some application developers prefer static storage such as arrays over complex heap structures. Examples for this pattern include wget and lighttpd. To ensure that this observation is not the result of false negatives, we manually inspected these two applications for undetected data structure implementations. As far as our evaluation goes, no data structures were missed by MEMPICK in these two applications.

Now let us focus our attention on the analysis of the erroneous classification reported by MEMPICK. The first example is a type misclassification in one of the linked list implementations in chromium. In this scenario MEMPICK reported a parent-pointer tree between the memory nodes. Browsing the source reveals the root of the error to be a programming decision. Nodes removed from the list never have their internal data cleared, nor are they freed until the end of the application. These unused memory links will stay resident in memory and confuse our shape analysis. A potential solution for this problem is a more advanced heap tracking mechanism with garbage collection. The latter would identify dead objects in memory and ensure that they are removed from the analysis. However we feel that this is not in the

scope of the current paper.

The other two type misclassifications both stem from composite data structures. Templated libraries such as STL make it possible for the programmer to build composite data structures like list-of-trees or list-of-lists. MEMPICK correctly identifies the data structure boundaries in situations where node types are mixed, but is unable to do so if both components have the same type, like dealing with list-of-lists. Without such boundaries, MEMPICK will evaluate the shape of the data structure as a whole. Intuitively, the resulting data structure still has a consistent shape, but features increased complexity. A combination of singly-linked lists turns into a child-pointer tree, while binary trees turn into ternary trees with the addition of the "root of sub-tree" pointer. This is also exactly what MEMPICK reports in these two scenarios. Pure shape analysis is not sufficiently expressive to distinguish between this pattern and regular child-pointer or ternary-trees, respectively. A reverse-engineer using MEMPICK can still identify this pattern with good confidence, by observing that the other partitions of the same type are classified as lists or trees.

Looking at the partition errors in table 3.5, the reader can notice that the vast majority belong to binary trees. We focus our attention on this class of errors first. For all misclassifications of this category, MEMPICK erroneously detects AVL balancedness instead of the weaker red-black or unbalanced properties. As presented previously in section 3.8 measuring the balancedness of a tree does carry uncertainty if the tree is too small. We confirmed that for each of the erroneous partitions, the tree contained no more than 7 nodes, a number too small to identify the difference between the two tree types. For all trees larger than this size our algorithm has an error rate of 0%.

Outside of the 3 main groups of errors, MEMPICK reports a few more misclassified partitions. Considering the total number of partitions reported across the 10 applications, these errors represent less than 1% and do not impact the overall analysis.

As part of evaluating, we also look at the analysis times required when processing these applications. We broke down the analysis times to different stages to identify potential problem areas within the analysis pipeline. We exclude the tracing component from this evaluation, since none of the proposed contributions relate to application tracing. The explicit tracing overhead can also be mitigated when combined with multi-path analysis. The KLEE family of multi-path analysis tools [21; 88; 89] is a prime example within the software engineering research community. Tools within the KLEE family emulate memory operations, by first looking up detailed information about the allocation site at the target address. The tracing within MEMPICK performs a similar look-up to identify the target heap object, while also performing a look-up on the value as-well. Thus, the desired tracing functionality could also be integrated within tools from the KLEE family with an additional 2X overhead in the worst case.

Table 3.6 presents the running time of the different analysis stages. The TypeGen stage includes type inference and the detection of the quiescent periods. The GraphGen stage represents graph generation, while the OverlayGen stage identifies al potential overlays. Finally the Classificaton stage is the time it takes to perform the final classification. Applications with limited heap usage finish within a matter of seconds as expected. Once the heap usage increases, so does the analysis time, especially for the TypeGen stage. This stage operates on raw traces and its execution time is unaffected by the semantics of the heap objects. This is highlighted within Lighttpd and Pachi, which make good use of heap memory, but few heap objects are members of high-level data structures. For these applications the bulk of the analysis is performed within the first stage, after which all non-desirable heap objects are purged from further analysis. Another particular application is Clang, where the OverlayGen stage is significantly more costly than the rest. This behavior is due to some heap objects featuring a large set of the pointer elements. Overlay identification requires testing an exponential number of pointer combinations, but for most data structures (except B-trees) the number of pointers is limited. Since we don't expect B-trees to come up often during analysis, this behavior can be considered an outlier and not the general case. Finally, for applications with heavy data structure usage, such as Tor and Wireshark, the execution time can increase to the range of minutes, but the total analysis time is still only around 30 minutes. These execution times suggest that the proposed methodology is well suited for the offline analysis of complex applications. Further optimizations can also be applied to reduce the analysis time within a production setting. For more detailed discussions about scalability, we refer the reader to section 3.11.

3.10.3 System code

One of the proposed use cases for MEMPICK was to analyze low-level system code for potentially vulnerable data structures. In this section we analyze the effectiveness of MEMPICK when dealing with this application class. MEMPICK relies on the PIN [71] framework for dynamic instrumentation, thus currently cannot analyze kernel-space code. However this does not mean that the mechanics behind MEMPICK cannot be applicable to system code. To overcome this technical limitation we leverage the FUSE project [126], which allows file system implementations to reside in user-space. Two major file-system implementations `NTFS-3g` and `ZFS-FUSE` are built on top of this framework on Linux. For this evaluation we choose two additional projects,`s3fs`, which allows mounting buckets from the S3 online storage service of Amazon and `sshfs`, which allows mounting remote folders via ssh.

Table 3.7 presents the overview of the results from MEMPICK when analyzing these four systems. The format is the same as the one used for the evaluation of real-world applications. For these four systems no typing errors were observed, only

partition errors where small red-black trees were mistakenly classified as being AVL trees. This type of error does not affect the ability of the reverse engineer to identify the underlying data structure since the results offer a comprehensive summary of all partitions, including the right classification. One peculiar detail is the lack of complex data structures for the two systems dealing with real file systems, `NTFS-3g` and `ZFS-FUSE`. No tree-like data structures related to inodes were discovered in the case of these two systems. By examining the intermediate results, we discover that MEMPICK correctly identifies the inode objects, but detects no direct pointer links between them. This discovery was confirmed by examining the underlying source code, which uses additional levels of indirection between inode objects. This programming pattern does not match our initial definition of homogeneous data structures. Future work may look into the discovery of heterogeneous data structures consisting of different object types. We conclude that MEMPICK was successful in analyzing these four systems and shows great promise in handling system code.

3.11 Complexity Analysis

In this section we analyze the computational complexity of the algorithms within MEMPICK, with the goal of proving that MEMPICK does not incur any hidden overhead that would impact its scalability. Our main argument in favor of dynamic analysis for the purpose of data structure detection is the inherent accuracy of run-time information that enables accuracy and scalability to coexist. To ensure that we meet our proposed performance goals, we analyze all components of our proposed approach, both from a theoretical and practical perspective. The research question we issue in this section is the following: "Is it possible to perform shape analysis using the same asymptotic complexity it takes to run basic test-suites for the application?". This research question stems from our assumption: the reverse engineer is capable of exercising the application with different inputs (as described in Section 3.2) and that he has access to powerful compute resources like clouds. The hope is that once the reverse engineer can execute the application itself, the analysis only involves a constant overhead that is independent of the application size and complexity. This constant overhead can be tackled by additional hardware resources if necessary. In the following, we analyze the complexity for five aspects of MEMPICK: executing the application, trace generation, type inference, graph generation and shape analysis. During the analysis we will use the following notations:

- N denotes the number of instructions executed and
- M denotes the number of heap objects generated by the application

3.11.1 Executing the application

We perform the dynamic analysis in MEMPICK by instrumenting the application under test and monitoring its behavior. This setup incurs two inherent sources of complexity that we consider in this section: the overhead of the instrumentation framework itself and the number of execution paths necessary to provide appropriate coverage. We show that neither of these two components hinders the goal proposed in our research question.

MEMPICK uses PIN as its instrumentation framework [71], which incurs a 4x overhead on the SPECint 2000 benchmark. In conclusion, the framework does not affect our complexity analysis, since our research question considers the native execution time as the baseline.

The second source of complexity we have to consider is the potential path explosion required for application coverage. In theory, an exponential number of possible execution paths exist for any application. We observed in Section 3.2) that full code coverage is not necessary for the purposes of data structure discovery. The core data structures of any application are used throughout the code, without concern for concrete execution details (considering realistic inputs). This is especially true for pointer-based data structures where life-times surpass function boundaries. Functional coverage is the primary metric for the purpose of MEMPICK and is typically provided by existing unit tests. While the size of unit tests can range up to several hundred inputs, the number is always bound to ensure the ability to finish testing in a reasonable time. Bounding the number of execution paths ensures that the asymptotic complexity metric is not affected, by our coverage requirements.

3.11.2 Trace generation

In this section we evaluate different techniques for monitoring and logging execution, to find an optimal solution that maintains the required linear complexity. MEMPICK generates traces as persistent and abstract representations of execution scenarios. The traces allow repeated analysis without having to deal with non-determinism during execution. They contain all events necessary for our analysis: heap object allocation and deallocation and writes into heap objects. The traces also abstract away low level details such as memory addresses into object identifiers to simplify analysis.

From an algorithmic perspective, tracing requires the system to instrument the individual instructions and generate short summaries whenever necessary. Whenever instructions deal with heap addresses, we abstract those into object identifiers and the corresponding offsets. Since an application may access various fields inside a memory object, each having a different offset, we cannot use a simple hash table to map addresses to object identifiers. The intuitive solution is to use an interval-tree like structure, where every allocation represents an interval. A look-up operation using any given memory address automatically results in both the object identifier

and offset. While this is an elegant solution to our problem, it does imply that for each memory write we incur an $O(logM)$ time look-up, where M is the number of heap objects. Thus the overall complexity of tracing is increased to $O(N * logM)$. An alternative solution is to use pointer tracking similar to Howard [119], which allows constant speed look-up for the root pointer of each memory operand. While pointer tracking can introduce a significant execution overhead, it does not change the asymptotic complexity itself. The overall tracing complexity thus remains $O(N)$.

3.11.3 Type inference

In this section we look into the complexity of the type inference algorithm presented in Section 3.5. In MEMPICK, type inference is the process of grouping objects into equivalence classes, called types. These equivalence classes do not necessarily correspond to the static types from source code. Each object is associated with a single type, while each type features a set of objects corresponding to it. MEMPICK leverages the definition of functional equivalence (as defined in Section 3.5) for this association. Whenever two objects are observed to be functionally equivalent, their types are merged. From an algorithmic perspective, this requires the two object sets to be merged as well as individual object types to be updated, resulting in a complexity linear with the smaller set size. Thus the worst case complexity occurs when equal sized sets are merged in a hierarchical fashion as presented in Figure 3.6. Considering the total number of objects to be M, this merging process operates with a worst case complexity of $O(M * logM)$. This is beyond the level of complexity we desire for our algorithm. In practice we found the algorithm to be scalable, thus we perform a more accurate complexity analysis to check for a potential over-approximation in the previous results.

For a more pragmatic approach on complexity analysis, we will follow the progression starting from a newly created object. When this object is first used as an operand for an instruction involved in data structure manipulation, its type is immediately merged with all other previous operands of the same instruction. We denote this new type as a first level type, as presented in Figure 3.6. Note, that every other object using the same instruction for the first time, will also be added to the same first level type. Thus, the potential number of first level types is bound by the different instructions that manipulate the data structure itself. These instructions are encapsulated within interface functions, and their numbers are directly proportional. In practice we never observed more than 10 interface functions for any data structure implementation. Since the type merging hierarchy from Figure 3.6 is independent for each individual data structure type, its height will be bound by the constant number of interface functions for that given type. This observation reduces the complexity of type merging to $O(M)$.

The overall complexity of type inference also needs to consider the time it takes to inspect all instructions in the trace file. However this is independent of the merging operations themselves from a complexity standpoint. The number of instructions

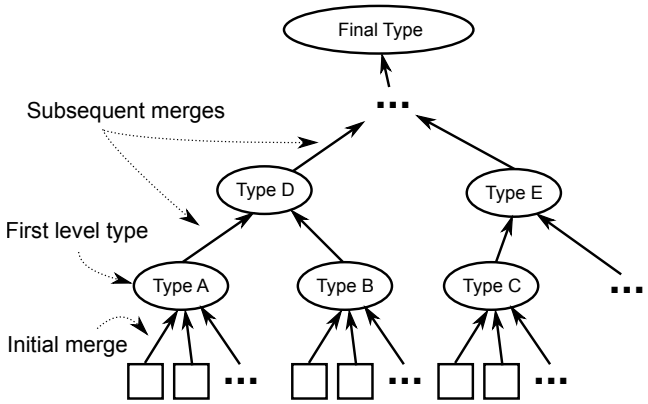


Figure 3.6: Visual representation of type merging process for objects of the same type. Objects are initialized with first level types, based on the first instruction manipulating them. Subsequent instructions trigger additional merge operations, generating new types. Finally all objects are merged together into one final type.

in the trace also dominates the number of objects created, thus the total complexity is maintained at $O(N)$.

3.11.4 Graph generation

Now, we consider the complexity of maintaining a graph representation of the heap in memory and generating periodic snapshots. Graph representation and storage is a well studied research field, that offers a wide variety of alternatives, depending on the problem requirements. In the case of MEMPICK, the graph contains a large number of nodes, potentially up to M and is highly dynamic, with potentially $O(N)$ operations performed. MEMPICK employs adjacency lists for its internal representation, since matrix-based solutions are prohibitive due to the potential node count. While the theoretical complexity of node and edge removal in adjacency lists is $O(E)$ (where E is the number of edges), we can bound the practical one by $O(D)$ where D is the maximum in/out-degree of each graph node.

This complexity is still beyond our requirements, thus we attempt to bound the value of D in practice, using the constraints of our problem. Our graphs do not represent generic shapes, but well formed data structures from within applications. Each data structure node is a structure, which contains a fixed number of pointers, that represent outgoing edges. In practice all the data structures we observed, except B-trees, featured up to 12 incoming and outgoing edges. The B-tree family of data structures can possibly contain up to thousands of pointers in each node. Even in this corner-case, D can be considered a constant value, albeit a large one. In scenarios where large B-trees may significantly impact performance, adjacency lists can be replaced by hash tables. The latter trades space overhead for additional performance guarantees. Thus we observe that choosing the appropriate graph representation

allows MEMPICK to maintain the graph in memory without incurring additional asymptotic complexity.

Besides the in-memory representation, MEMPICK is also required to provide periodic graph snapshots at quiescent periods. While generating an individual snapshot depends on the graph size ($O(M+E)$), the total number of quiescent periods is also trace dependent ($O(N)$). Thus we observe that it is not scalable to generate all possible graph snapshots. To mitigate this issue, the system selects a random sample containing K snapshots that are forwarded to graph analysis. The reverse engineer can control this number, based on the level of confidence desired. While in theory the accuracy of MEMPICK may be affected, in practice we did not observe such issues. The evaluation in Section 3.10.2 was performed using up to 100 snapshots for each data structure instance and none of the classification errors resulted from this limitations.

Combining all components of graph generation results in a complexity of $O(N+K*(M+E))$. The final formula is controlled by the value of K chosen by the reverse engineer. Constant values ensure that the complexity is still within the desired $O(N)$ class. Our evaluation shows that detection accuracy is not affected by this decision, but for the purpose of precise representation we will keep K as a variable for the rest of the analysis.

3.11.5 Shape analysis

Finally, we analyze the complexity of the classification algorithm which operates on the generated snapshots to produce the final results. This algorithm consists of two components, namely overlay identification and rule-based classification of the overlays.

In Section 3.7.1, we described the notion of minimal pointer sets which define the possible overlays in a given data structure. Since this problem is analogous to the maximal set and set cover problems, the potential number of sets can be considered exponential in the potential pointer count. MEMPICK validates each candidate pointer set, by analyzing the connectivity in the K snapshots generated in the previous step. This process is performed in $O(K * (M + E))$ for each pointer set highlighting the necessity to deal with the exponential set count. Previously we observed that the number of potential pointers, D, is limited to a constant in practice, thus ensuring a constant number of potential sets.

The final shape analysis requires MEMPICK to analyze the in/out degrees for the discovered overlays. Classification itself is performed in a constant time, once the statistics are gathered for each snapshot. The resulting time complexity is $O(K * (M + E))$, identical to the result for overlay detection.

3.11.6 Summary of complexity analysis

In summary, by adding together all steps of MEMPICK, the total time complexity of data structure detection is $O(N + K * (M + E))$. Our evaluation has shown that choosing a reasonable constant number for K does not impact the accuracy of MEMPICK regardless of the application under analysis. As such the time complexity formula is simplified to $O(N)$, since the trace size dominates both the potential number of memory objects and pointers. Going back to our research question, we have shown that it is possible to perform data structure detection using the same time complexity required for regular application execution. This ensures the scalability of our solution even for complex applications. While the constant overhead incurred is not negligible, hardware and parallel execution may be used in the future to mitigate it. Even now, the current implementation of MEMPICK finished the analysis of each application from our evaluation (Section 3.10.2) within one hour on a 2.6 GHz dual-core machine. This result and our complexity analysis proves that MEMPICK is applicable for the overnight analysis of these application classes.

3.12 Limitations and Future Work

In this work we aim to detect and classify heap based data structures using shape analysis applied to the memory graph. Applications use memory allocators to manage heap objects, a facility instrumented in MEMPICK to maintain an accurate representation of the memory graph. While most applications employ system allocation routines like `malloc()` or `free()`, some applications implement custom memory allocators for performance benefits. In the latter scenario MEMPICK needs to be made aware of the custom memory allocators in use by the application. While this information is not readily available in stripped binaries, the approach by Chen et al. [26] is straightforward to adapt for the requirements of MEMPICK.

The shape analysis of the memory graph in MEMPICK is based on a set of simple, but stringent rules geared towards edge counts. This classification mechanism assumes the ability to discerning relevant and irrelevant edges in the memory graph via some typing information. Our evaluation shows that the type inference engine designed for MEMPICK can meet this requirement in practice, but some theoretical corner cases still exist. Typeless pointers, unions or inner structs could confuse our current solution in theory. For the future we propose the fusion of multiple typing information sources, such as Howard [119] or static analyses [13; 108; 110] to limit potential false positives.

In addition, we focus on data structures that can be classified based solely on their shape, and not the contents or algorithms used to handle them. For example, we cannot distinguish binary search trees from the generic binary trees.

A natural extension of MEMPICK is the functional analysis of data structures. MEMPICK currently identifies all the instructions involved in the internal operations of the data structure, but is unable to reason about them. The reverse engi-

neering value would be expanded by labeling the instructions with their functional purpose(*insertion, deletion*). We believe that the existing shape analysis results significantly reduce the space of possible operations, enabling a robust and intuitive functional classification. This extension will allow reverse engineers to quickly identify code related to the known semantics of data structures and focus their attention on application logic instead.

3.13 Related Work

Recovery of data structures is relevant to the fields of shape analysis and reverse engineering. While shape analysis aims to prove properties of data structures (e.g., that a graph is acyclic), reverse engineering techniques observe how a binary uses memory, and based on that identify properties of the underlying data structures. In this section, we summarize the existing approaches and their relation to MEMPICK.

Shape analysis. Shape analysis [50; 112; 79; 17; 138] is a static analysis technique that discovers and verifies properties of linked, dynamically allocated data structures. It is typically used at compile time to find software bugs or to verify high-level correctness properties of programs. Although the method is powerful, it is also provably undecidable, and so conservative. It has not been widely adopted.

Low-level data structure identification. The most common approaches to low-level data structure detection, i.e., primitive types, `structs` or arrays, are based on static analysis techniques like value set analysis [13], aggregate structure identification [108] and combinations thereof [110]. Some recent approaches such as Rewards [86], Howard [119], and TIE [85], have resorted to dynamic analysis to overcome the limitations of static analysis. Even though they achieve high accuracy, they cannot provide any information about high-level data structures, such as lists or trees. MEMPICK is thus complementary to them.

High-level data structure identification. The most relevant to our work are approaches that dynamically detect high-level data structures, such as Raman et al. [109], Laika [34], DDT [74], and White et al. [133].

Raman et al. [109] focus on profiling recursive data structures. The authors introduce the notion of a shape graph, that tracks how a collection of objects of the same type evolves throughout the execution. MEMPICK's memory graph extends the shape graphs to facilitate data structure detection, which is beyond the scope of the profiler [109].

Laika [34] recovers data structures during execution. First, it identifies potential pointers in the memory dump—based on whether the contents of 4 byte words look like a valid pointer—and then uses them to estimate object positions and sizes. Initially, it assumes an object to start at the address pointed to and to end at the next object in memory. It then converts the objects from raw bytes to sequences of block types (e.g., a value that points into the heap is probably a pointer, a null terminated sequence of ASCII characters is probably a string, and so on). Finally, it

detects similar objects by clustering objects with similar sequences of block types. In this way, Laika detects lists and other abstract data types. However, the detection is imprecise, and insufficient for debugging or reverse engineering. The authors are aware of this and use Laika instead to estimate the similarity of malware. Similarly to Laika, Polishchuk et al. [104], SigGraph [87], and MAS [35], are all concerned with identifying data structures in memory dumps. However, they all rely on the type related information or debug symbol tables.

White et al. [133] propose an alternative to shape analysis, by focusing the analysis on the patterns in data structure operations. They label instruction groups based on the local changes observed in the pointer graph. Finally they merge the label information from all instruction groups to form a final candidate classification. The main issue with this approach lies in the complexity of the underlying model, which requires a repository of manually defined templates to perform classification. The authors also require source code access to extract typing information for the pointer graph. Finally, their evaluation is limited to very simple applications which use a single data structure internally. With MEMPICK we have shown that shape analysis can provide the necessary accuracy, while benefiting from simple and intuitive models. While MEMPICK does not yet support the analysis of data structure operations, we strongly believe, that the result of the shape analysis is highly valuable to limit the search space of such analysis.

Guo et al. [59] propose an algorithm to dynamically infer abstract types. The basic idea is that a run-time interaction among (primitive) values indicate that they have the same type, so their abstract types are unified. This approach groups together objects that are classified together, e.g., array indices, counts or memory addresses. MEMPICK's approach to type identification (Section 3.5) is less generic, but also simpler and specifically tailored to our needs.

Currently, the most advanced approach to the data structure detection problem is DDT [74]. DDT uses invariant information extracted using Daikon [43]. This allows DDT to go beyond shape information and to refine its classification based on content information. Unfortunately the same invariant detection also imposes additional assumptions on the system, reducing its flexibility. For one, DDT relies on well-structured detect interface functions which encapsulate *all* operations performed on data structures. The distinction is very strict: the system assumes that an application never accesses any links between heap objects, while the interface functions never modify the contents they store in the data structures. Thus, the applicability of DDT is limited when due to compiler optimizations, the interface functions are inlined, their calling conventions do not follow the standard ones, or when a program simply uses data structures defined with macros or some less strict interfaces (e.g., `queue.h`). In the absence of inlining, DDT works well with popular and mature libraries, such as the C++ Standard Template Library (STL) or the GNOME C-based GLib, but it is unclear what accuracy it would achieve for custom implementations of data structures (let alone malware). MEMPICK does not make any assumptions about the structure of the code implementing the operations on data structures, so

it has no problems analyzing applications that use `queue.h`, say. Additionally, DDT does not address the problem of the auxiliary overlays in data structures. For each data structure type, it relies on a *graph invariant* that summarizes its basic shape. For example, one of the invariants specifies that "each node in a binary tree will contain edges to at most two other nodes". However, this assumption does not always hold in practice.

3.14 Conclusion

In this paper, we presented MEMPICK, a set of techniques to detect complicated pointer structures in stripped C/C++ binaries. MEMPICK works solely on the basis of shape analysis. The drawback of such an approach is that it will only detect data structures that can be distinguished by their shape. On the other hand, we showed that MEMPICK is impervious to compiler optimizations such as inlining and accurately detects the overall data structure even if it is composed of multiple overlapping substructures. We evaluated MEMPICK first on a set of 16 common libraries and then on a diverse set of ten real-world applications. In both cases, the accuracy of the data structure detection was high, and the number of false positives quite low. In conclusion, we believe that MEMPICK will be powerful tool in the hands of reverse engineers.

Acknowledgment

This work is supported by the European Research Council through project ERC-2010-StG 259108-ROSETTA, the EU FP7 SysSec Network of Excellence and by the Microsoft Research PhD Scholarship Programme through the project MRL 2011-049.

Table 3.5: MEMPICK's evaluation across 10 real-world applications. #T is the number of unique data structures belonging to the given type, #MT is the number of type misclassifications, #P is the number of partitions belonging to the given type, #MP is the number of partition misclassification

Application	Type	#T	#MT	#P	#MP
chromium	slist	16	0	303	0
	dlist	5	0	24	0
	list of lists	1	1	8	8
	n-ary tree	1	0	16	0
	n-ary tree	1	0	2	0
	slist + graph	1	0	169	2
	graph	2	0	10	1
clang	slist	3	1	5	1
	dlist	5	0	8	0
	RB tree	1	0	6	2
	graph	4	0	13	0
inkscape	slist	9	0	186	0
	dlist	5	0	14	0
	RB tree	1	0	7	4
	tree of trees	1	0	5	0
	n-ary tree	1	0	28	0
	slist + graph	1	0	13	0
	graph	1	0	1	0
lighttpd	slist	2	0	2	0
	dlist	1	0	1	0
	binary tree	1	0	1	0
pachi	n-ary tree	1	0	1	0
povray	slist	9	0	36	0
	dlist	3	0	66	2
	RB tree	1	0	1	0
	n-ary tree	1	0	17	0
	n-ary tree	1	0	16	1
	slist + graph	1	0	12	0
quagga	slist	2	0	7	0
	dlist	5	0	8	0
	binary tree	1	0	4	2
tor	slist	12	0	413	4
	graph	1	0	1	0
wget	slist	3	0	8	0
	dlist	1	0	6	0
	slist + graph	1	0	13	0
wireshark	slist	3	0	99	0
	dlist	1	0	1071	0
	binary tree	1	0	1	0
	n-ary tree	1	0	3	0
	RB tree	1	0	95	47
	AVL tree	1	0	2	0
	slist + graph	1	0	12	0
	graph	1	0	1	0

Table 3.6: MEMPICK's analysis time evaluation across 10 real-world applications. Averaged across 3 runs. Measured in seconds. TypeGen includes the type inference and quiescent period detection. GraphGen involves graph generation, while OverlayGen includes the overlay identification. Classification includes the remaining classification steps to get the final results.

Application	TypeGen	GraphGen	OverlayGen	Classification
chromium	2s	<1s	<1s	2s
clang	4s	<1s	178s	4s
inkscape	8s	4s	5s	9s
lighttpd	23s	<1s	<1s	<1s
pachi	21s	<1s	<1s	<1s
povray	38s	1s	1s	4s
quagga	<1s	<1s	<1s	<1s
tor	848s	542s	42s	361s
wget	8s	4s	1s	3s
wireshark	160s	1030s	247s	453s

Table 3.7: MEMPICK's evaluation for the 4 FUSE-based file systems. #T is the number of unique data structures belonging to the given type, #MT is the number of type misclassifications, #P is the number of partitions belonging to the given type, #MP is the number of partition misclassification

Application	Type	#T	#MT	#P	#MP
NTFS-3g	slist	1	0	1	0
	dlist	2	0	2	0
	slist + graph	1	0	24	0
ZFS-FUSE	slist	2	0	2	0
	dlist	1	0	1199	0
s3fs	slist	3	0	245	0
	dlist	1	0	1	0
	RB tree	1	0	62	37
	n-ary tree	1	0	19	0
	slist + graph	1	0	13	0
sshfs	dlist	2	0	16	0

4

ShrinkWrap: VTable Protection without Loose Ends

Abstract

As VTable hijacking becomes the primary mode of exploitation against modern browsers, protecting said VTables has recently become a prime research interest. While multiple source- and binary-based solutions for protecting VTables have been proposed already, we found that in practice they are too conservative, which allows determined attackers to circumvent them. In this paper we delve into the design of C++ VTables and match that knowledge against the now industry standard protection scheme of VTV. We propose an end-to-end design that significantly refines VTV, to offer a provably optimal protection scheme. As we build on top of VTV, we preserve all of its advantages in terms of software compatibility and overhead. Thus, our proposed design comes "for free" for any user today. Besides the design we propose a testing methodology, which can be used by future developers to validate their implementations. We evaluated our protection scheme on Google Chrome and show that no compatibility issues were introduced, while overhead is also unchanged compared to the baseline of VTV.

4.1 Introduction

C++ is a popular, fast, object-oriented (OO) language used to develop some of the most popular Web browsers, including Chrome and Mozilla. Due to their popularity, size and complexity, applications developed in C++ are frequently targeted by attackers. Despite advances in software security, like the introduction of data-execution prevention [8], stack-smashing protection [33], and address-space layout randomization [103], their exploitation is still possible. New techniques involving information leaks [121] and return-oriented programming [116] are employed to bypass protection mechanisms and perform arbitrary code execution attacks.

One of the features of C++ applications targeted by attackers are *virtual function tables*, or *VTables*. OO languages support run-time method binding, i.e., determining the method to be called based on the run-time type of an object, instead of the static type of the pointer pointing to that object. Modern compilers typically provide this functionality through *VTables*, which provide an efficient way to call the correct method at run time. Unfortunately, VTables are based on indirect calls, which is what makes them a prominent targets for hijacking the control flow of a program.

To prevent such control-hijacking attacks, the research community has turned to *control-flow integrity* (CFI). First conceived in 2005 [3], CFI has seen a long line of followers and variants since [139; 141; 99]. CFI strives to constrain the control flow of a program to its statically-determined control-flow graph (CFG) as strictly as possible. In principle, CFI can be very effective in preventing a wide-range of attacks. In recent times however, we bear witness to a cat-and-mouse game, where each new CFI technique is immediately attacked and bypassed. Earlier works have shown that attackers can bypass loose CFI mechanisms [54], so follow-up works tried to exploit source code information [99] and VTables semantics [72; 25; 9] to make CFI more fine-grained. A very recent work has shown that the above approaches still leave programs vulnerable and argue that unless you correctly extract C++ semantics from source code they will remain vulnerable in the future [113].

This paper aims to provide the final say on VTable protection, by tightly constraining virtual function pointers (vfp) to the VTables corresponding to the classes intended by the programmer—as defined by the semantics of the C++ language. We begin by examining a recent compiler-based CFI approach, namely VTV [127], and evaluate it to determine whether its own vfp restrictions are accurate.[1] We proceed by extending VTV to apply even tighter restrictions to the available VTable targets and argue that our approach is optimal. More precisely, we aim at offering the best protection possible to vfps in a context insensitive fashion. We build our solution into VTV and evaluate it by means of a new framework for testing it. Using the framework, we experimentally show that it is the strictest access policy possible for VTables without breaking legitimate code. Last but not least, our solution is *faster* than the original VTV implementation.

[1] VTV is now a standard compiler option also used in production systems.

Our investigation exposes *three* weaknesses of existing schemes. First, we find that existing solutions fail to precisely identify the object types associated with a virtual call-site, even in the presence of source code. Second, we find that even state-of-the-art solutions, like VTV, handle multiple inheritance over-permissively. Normally, every class has its own VTable and base classes contain all the VTables of their subclasses. When a class C inherits from multiple classes, VTV extends the VTables of its base classes to include, and thus share, all entries in their individual VTables. I.e., "sibling" classes share VTable entries. This is another example where control-flow integrity is loosely enforced. Finally, we identify a fundamental error in the assumptions made by other solutions. Previous approaches operate on the premise that allowable control-flow transfers at call-sites (i.e., where a method of an object is invoked) can be determined solely based on the type of the object pointer involved. We show that this assumption is false and more information must be extracted from the call-site to reach optimal protection.

We use our observations to design a new VTable-protection scheme that uses information available during compilation to extract the most restrictive set of VTables that should accessible at a virtual call-site within the code. We implement this enhanced, fine-grained design on top of VTV and evaluate it by creating a testing framework that exhaustively explores all possible combinations of class inheritance and method invocation to demonstrate that our technique provides the best possible defense for VTables. We also experimentally test our approach by compiling and running the Chrome Browser to demonstrate that our modifications do not break complex, real-life applications. The evaluation of our prototype also shows that our scheme is faster than the original VTV scheme.

We summarize our contributions below:

- We identify limitations in the design and implementation of current VTable protection schemes, including the primary industrial implementation, VTV. (Section 4.2.2)

- We identify key design decisions that should be accounted for, when dealing with VTable protection. This also includes a definition of optimal (minimal) VTables sets that should be accesible at each point in the program. (Section 4.3)

- We develop a practical testing methodology to evaluate VTable protection schemes and to highlight potential limitations.(Section 4.5.1)

- We implement a prototype of the proposed protection scheme and evaluate it on a large, complex real-world application, the Chrome, browser, in terms of security *and* speed. (Section 4.5.2)

4.2 VTable Protection Today

In this section, we discuss VTables and their protection in current solution, as well as the reasons why such defenses are not as tight as they should be.

4.2.1 C++ dynamic dispatching

Function polymorphism in object oriented languages like C++ needs a way to dynamically resolve the appropriate method implementation based on the dynamic type of the object. For instance, if B and C are subclasses of A, and both implement a method `f()`, we can initialize any reference to A with an object of either type B or C. However, when we now call the method `f()`, we execute either `B.f()` or `C.f()`, depending on the dynamic type. The typical solution to this problem is to group all methods of a particular class into a table of function pointers, called VTable in C++. Subclasses extend the VTable of their base class with new entries for newly defined methods, while previously defined methods feature updated function pointer entries. During object construction, a pointer towards the appropriate VTable is stored within the object. This object, together with the VTable it points to, allows the compiler to select the appropriate polymorphic method variant, irrespective of the compile-time type.

C++ also features complex inheritance strategies, such as multiple and virtual inheritance, that affect VTable usage. For example, assume that class *B* simultaneously inherits from multiple base classes *A1, A2, ..., An*. The VTable of *B* can only extend the VTable of the *primary* base class for *B*. Otherwise entries in the VTable would have to overlap on the same offset. To allow using the class *B* in place of all of its parents, *secondary* VTables are associated with it, corresponding to each non-primary base. These secondary VTables are also inherited in all of the subclasses of *B*. Virtual inheritance is a necessary side-effect of supporting multiple inheritance. The latter allows the same base class to be inherited multiple times via different inheritance chains. This is potentially undesirable behavior. Virtual inheritance solves the problem, by ensuring that a given base class is inherited a single time in any further subclasses. Virtually inherited base classes also trigger the generation of their own secondary VTables to be used when accessing methods from this particular base. For more details about VTable interaction with inheritance, the reader can refer to the C++ ABI documentation [1].

While each compiler generates its own code for supporting VTables, a common approach is to store at the beginning of an allocated object a pointer to the object's VTable. Normally, VTables themselves are stored in read-only memory to prevent tampering by attackers. However, C++ objects can be allocated on the stack and the heap, which are both writable. Therefore, the pointer that points to the VTable can be overwritten by leveraging a software bug (like a buffer overflows or user-after-free bug [5]), for instance to make it point to VTable-like data the attacker controls.

4.2.2 VTable integrity and limitations

It is evident that protecting VTable pointers will make software exploitation dramatically harder. As a result, recent security conferences abound with publications on how to protect VTables. Although all these proposals apply some notion of CFI, some are VTable-agnostic [139; 141], while others target VTables specifically [127; 72; 25; 9]. In the following we focus on the second category, as their understanding of VTable semantics allows the protection to be more refined than generic approaches.

Since all recent VTable protection schemes [127; 72; 25; 9] share a similar architecture, we map them onto the following model to analyze their strengths and weaknesses.

1. Statically search for VTable based call-sites.
2. Statically generate VTable sets that could be associated with each class/call-site.
3. Statically identify the class type used at each call-site.
4. Enforce that run-time VTables are part of the statically inferred set at each call-site.

In short, each protection scheme aims to associate a set of valid VTables to each particular call-site. By enforcing that the run-time VTable belongs to the statically generated set, it aims to limit the influence of the attacker on the control-flow. The sets should contain all possible VTables that could be used at the given call-site. In order to avoid having a new set for every call-site in the program, the sets are typically grouped together, based on type information. The intuition is that call-sites with the same static object type have access to the same VTables.

Recent binary- and source-based solutions have been successful at solving the first and fourth points in this model, and we do not cover them much in this paper. Instead, we focus on the second and third points, where we identified limitations in all existing solutions.

Generating VTable sets

VTable sets contain all VTables that a particular call-site can legitimately target. The best way to generate them is through analyzing the class hierarchy, since it defines how valid C++ code interacts with VTables.

Binary level Current binary based approaches are limited by the information that they can extract for a particular call-site. Since type information is unavailable in closed-source programs, binary protection schemes cannot differentiate between the legitimate targets for different call-sites. As a result, they typically use a single VTable set that contains all VTables accumulated from the binary. While this stops many existing exploits, attackers are still able to corrupt the program flow. For example, a call-site calling a virtual method of 0 arguments can be used to call into

a method with 3 arguments. Attackers can exploit this pattern to perform stack pivoting on Windows based systems as shown by Göktaş et al. [54]. Prakash et al. [9] try to extract limited semantics from the call-site to support more specialized VTable sets. However, the authors admit that even their advanced policies lead to VTable sets being larger by a factor of 2X, compared to the existing source-based VTable protection in GCC.

Source level VTV [127] takes a class-based approach for generating the sets. It associates a VTable set with each class which includes all its VTables and the VTables of its subclasses. This is intuitive, as all subclasses can be used at the call-sites of their base classes. SafeDispatch [72] provides two alternatives for generated VTables: (a) the VTV scheme, and (b) method-based VTable sets. In case of the latter each virtual method is associated with a VTable set of its own, based on method overloading in the subclasses. Although this alternative reduces the run-time overhead, according to the authors it is weaker or equivalent to the VTV approach and we will not focus on it in the remainder of this paper.

While VTV's solution of adding all VTables of subclasses to the set is intuitive at first glance, it fails to arrive at the right set in case of multiple inheritance. In this case,classes inherit completely unrelated functionalities using multiple unrelated VTables. Code at the call-site assumes that the appropriate casting mechanisms have been applied to select the desired VTable of the object before making the call. However, VTV allows attackers to inject a different (and mismatching) VTable of the same class at the call-site, undermining its semantics. We present the security impact of this limitation on the Chrome browser in Section 4.5.2 (with callsites erroneously giving access to thousands of VTables).

In this paper we propose an in-depth analysis of class hierarchies and VTable generation policies. Based on the analysis, we then propose a VTable set generation policy, which does not suffer from false positives and still supports all valid C++ semantics (Section 4.3.2).

Call-site type inference

While this step is typically left as an implementation detail in previous papers [127; 72; 25; 9], we believe that it should be an integral part of the design and evaluation of VTable protection. Since type inference is the key for associating call-sites with particular VTable sets, this component has a direct influence on the number of allowable VTables. Even if the VTable sets are generated to be optimal, it is enough to associate the wrong (overly conservative) set at a particular call-site to increase the attack surface. For example, imagine a class hierarchy, similar to Java with a common root class `Object`. Since all other classes are based on `Object`, conservatively associating a call-site with the root class allows an attacker to use of all VTables within the system, which can lead to a significant and undesirable attack surface increase.

Since type inference is still very difficult to perform at the level of binaries, most

binary solutions forego call-site type inference entirely and limit themselves to a single VTable set that they associate with each call-site. As mentioned before, doing so allows attackers to leverage all VTables within the system at every virtual call-site, which might be enough for future exploits.

We expected source based solutions to have solved the problem of type inference completely, as they have access to the underlying code, but this turned out to be wrong. For instance, VTV [127] turns out to be overly conservative in this stage, and as a result not nearly as effective in validating VTable pointers as it could be. Specifically, we observed that, in the case of multiple and virtual inheritance, the type inference scheme in VTV is prone to associate call-sites with base classes of the type specified in the source. We discuss the problem of precise call-site type inference in detail and provide a compiler-agnostic solution in Section 4.3.1.

4.3 ShrinkWrapping the VTables

4.3.1 Precise call-site type inference

As precise VTable protection relies on associating the appropriate VTable set with each call-site, call-site type inference is crucial. If type inference is too conservative, the call-site might be associated with a super-class instead, allowing the use of VTables with no relationship to the given call-site. We found that VTV [127], currently the state-of-the-art in VTable protection, suffers from overly conservative type inference and later in Section 4.5.2, we highlight the impact of VTV's conservative nature when protecting call-sites in complex programs like Google Chrome. In the following we analyze the root cause of type inference issues within GCC. We also map our observations to other compiler frameworks to suggest a generic design for future VTable protection implementations.

The source of the conservativeness stems from the fact that the core of a typical C/C++ compiler is built to handle a wide range of language front-ends, and thus oblivious to language specific features, such as C++ VTables. It is the responsibility of the C++ front-end to transform VTable-based method calls into traditional calls. This process involves implicitly casting the object to its base class which explicitly implements the desired method. When performing instrumentation within the core of the compiler, this pattern is impossible to separate from explicit casts and field accesses. For example, in Figure 4.1 a call-site using a pointer of type *C* accessing a virtual method inherited from *B* will be associated with type *B* or even *A2* due to a bug in the VTV implementation. This inherently enables access to a larger number of VTables than desired by the programmer. We discovered these issues, while analyzing the code in GCC, but the problem applies to most compiler frameworks. For example, the core of Clang uses the language agnostic LLVM intermediate representation, which also lacks type information for virtual method call-sites as mentioned by Jang et. al. [72]. Likewise, the Microsoft compiler also uses separate language-specific front-ends C1 (C) and C1XX (C++) to parse the code and transform it into

an intermediate representation processed by the C2 back-end, and while compiler internals are not known, it seems likely that it loses virtual method semantics at the level of the back-end as a result of normalization with raw C code. With this analysis we hope to draw attention to the issue of type inference in C++, so that future VTable defenses (regardless of the targeted compiler) will use *precise* call-site type inference.

As a baseline solution, we propose using the earliest possible stage within the C++ front-end and parser to perform type inference, and propagate the information to the instrumentation code via internal compiler annotations. This allows the instrumentation to reside either in the front-end or the core of the compiler, without affecting the precision of type inference. In our setup we annotate the access to the VTable pointer itself (as it is generated in the front-end) and transform the annotation into a VTable check at instrumentation time. It is even possible to include further analysis in the compiler to infer a more restrictive type to associate with the call-site based the static pointer-tracking, but we leave this up for future work.

4.3.2 Legitimate VTable targets

As described in Section 4.2.2, VTV [127] uses a coarse definition of allowable VTable sets for each call-site. If class `B` is a sub-class of `A`, then all virtual call-sites using the latter type are allowed to use any of the VTables found within `B`. However, Section 4.2.1 showed that multiple and virtual inheritance can result in classes having a large range of VTables. In this case some of the VTables in class `B` are not inherited from `A`, and should thus be inaccessible at a call-site using type `A`. In Section 6.9, we show that VTV inadvertently allows some call sites in the Chrome browser access to thousands of VTables. We introduce a pair of concepts to model the relationships between VTables: the type of a VTable and the parent relationship between a pair of VTables. They form the basis for generating provably optimal VTable sets for each call-site.

Concepts First, we define the *type* of a VTable to be the base class of the object responsible for triggering the generation of this particular VTable. For example, the *primary* VTable of a class has the same type as the class. In the case of multiple inheritance, every *secondary* base (including inherited ones) generates its own VTable, with the given class as its type. An example of the type association is presented in Figure 4.1. Second, we define a *parent relationship* between VTables of different classes.

A VTable `X` in class `A` is a parent of VTable `Y` from class `B` if and only if:

- A==B or class `A` is a base class of class `B`.
- VTable `Y` is inherited from class `A`.
- VTable `Y` matches or extends VTable `X`.

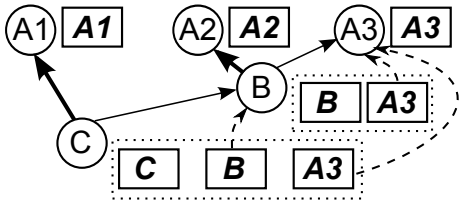


Figure 4.1: Example class hierarchy. The classes are represented by circles. The solid arrows show parent relationships between classes, the thicker ones signaling the primary parent. The VTables of each class are represented by the rectangles next to it. The dashed arrows signal the class from which a particular VTable was inherited. The text in each rectangle is the type associated with the corresponding VTable, based on the inheritance.

Analogously VTable `Y` is a descendant of VTable `X` if VTable `X` is a parent of VTable `Y`.

VTable extension is defined in C++ as generating a new VTable that starts out with the same layout as the original one, but with additional entries appended. VTable extension allows efficient type-casting without the need of generating additional VTables. An extended VTable can always be used in place of the original, as the relevant part matches in layout.

To identify when a VTable extends another, we take a look at inheritance rules for both multiple and virtual inheritance. The primary VTable of a class always extends the primary VTable of its first non-virtual base class. Extension continues transitively as long as non-virtual inheritance is involved. The example in Figure 4.2 shows the parent relationships between the VTables introduced in Figure 4.1. Because class `A2` is the primary base class of `B`, its VTable is extended as the primary VTable of `B`, leading to a parent relationship between the two VTables.

In case of virtual inheritance things get more complicated. Virtual base classes are only inherited once in sub-classes. As a result the VTable of this base is only extended a single time, even in the face of diamond inheritance as presented in Figure 4.3. While the VTables of both `B1` and `B2` extend the VTable of `A`, this property is not transitively propagated into the sub-class `C`, only the primary VTable of `C` extends the one inherited from `A`. In the second VTable of class `C` the entries corresponding to `A` are eliminated from the VTable. As a result, in the face of virtual inheritance, explicit analysis of the VTable content is necessary to identify extension between two VTables.

Usage Based on this definition of the parent relationship, it is intuitive that only descendant VTables can be used in place of their parents at any particular virtual method call-site. The layouts of the descendants always match the layout expected at the call-site, resulting in a successful method invocation. Any VTable which is not a descendant should *not* be allowed at the call-site, as it was either inherited from a different, unrelated base class, or its layout does not match the expectations

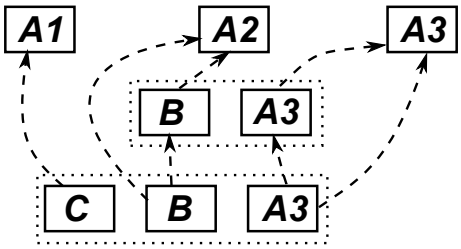


Figure 4.2: Visualization of the parent relationship of VTables for the class hierarchy from Figure 4.1. This figure leaves out the classes, preserving only the VTables. The dashed arrows represent the parent relationship between two VTables. The VTable in `A2` is a parent for both VTables of type `B`, since `A2` is the primary parent of `B`, and the primary VTables of the latter is an extension of the VTable inherited from `A2`. The same between the VTable of type `C` and `A1`.

at the call-site. These properties make the parent relationship the perfect basis for generating allowable VTable sets, based on C++ class semantics.

4.4 Stronger VTable Protection

We introduce stronger VTable protection through (i) a simple extension to VTV, and (ii) an optimal solution.

4.4.1 An extension to VTV

As a first step toward better security, we use the concepts presented above to redefine the restrictions enforced by VTV [127] and increase the strictness of its protection. The existing implementation traverses the class hierarchy at compile-time to identify targeted VTable sets for each class within the system. We extend this mechanism by a set of additional restrictions to limit the contents of this set further, while maintaining full compatibility with all C++ semantics.

We propose the following simple policy for VTable sets at a call-site corresponding to a particular class:

- All VTables of the class are part of the set.
- All descendant VTables of the above are also part of the set.

This policy ensures that VTables in the sub-classes, with no relation to the given class, are never added to the set.

Implementing the policy is straightforward, only requiring information about VTable type and parent relationships. These are extracted from the class hierarchy, based on their definition from above. As with the call-site type inference, we believe that the proper testing methodology would have highlighted the limitations of the existing design earlier in the development process.

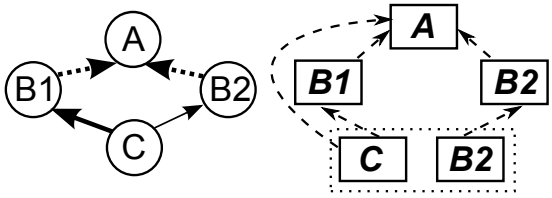


Figure 4.3: Example of diamond virtual-inheritance, where by courtesy of the virtual inheritance only a single copy of the top class (`A`) is inherited within `C`. While the primary VTable of class `B2` does include all entries corresponding to parent `A`, when this VTable is inherited into `C`, the entries are cleared out. Irrelevant of the type-casting chain used to convert an object of type `C` into type `A`, the primary VTable of `C` will always be used to access the corresponding methods. The parent relationship follows this semantic, since only one of the VTables within class `C` has the primary VTable of `A` as its parent.

4.4.2 Optimal VTable protection

While the previous solution is intuitively strong, it is limited by a core design decision, common to both VTV [127] and SafeDispatch [72]. Both these papers assume that VTable-level protection is based on a *single* piece of type information, particular to the call-site. In contrast, when a call is performed using a virtual method, the compiler knows *two* things: the type of the object on which the method is called, and the particular VTable of this object, where the method can be found (represented by the type of the VTable for example). When limiting the protection scheme to a single type being associated with the call-site, information is inadvertently lost, degrading the precision of the protection. We propose leveraging both pieces of information, which enables a fine-grained and provably optimal VTable protection scheme. The new design is defined as follows:

- Each call-site is represented using a class-type and VTable pair.
- The set is initialized with only the call-site VTable.
- All descendant VTables of the above are also added to the set.

This scheme is optimal, since the VTable sets only include the entries mandated by the C++ semantics. This is guaranteed by the definition of the parent relationship from Section 4.3.2. Every entry from in the set can potentially be used at the call-site, via a valid type-casting chain. Figure 4.4 shows a comparison between the original VTV, the extension we presented in Section 4.4.1 and our new fine-grained solution (see Section 6.9 for a quantitative analysis).

Implementation-wise this new scheme can also be added on top of VTV, while preserving much of the core code intact. Call-site type inference needs to be extended to also provide information about the VTable in use, while the VTable sets are also kept track of with finer granularity. The end result is a slight increase in the code size to support the newly defined additional VTable sets. Notice that this policy results in smaller individual set sizes, which leads to a reduction in run-time overhead.

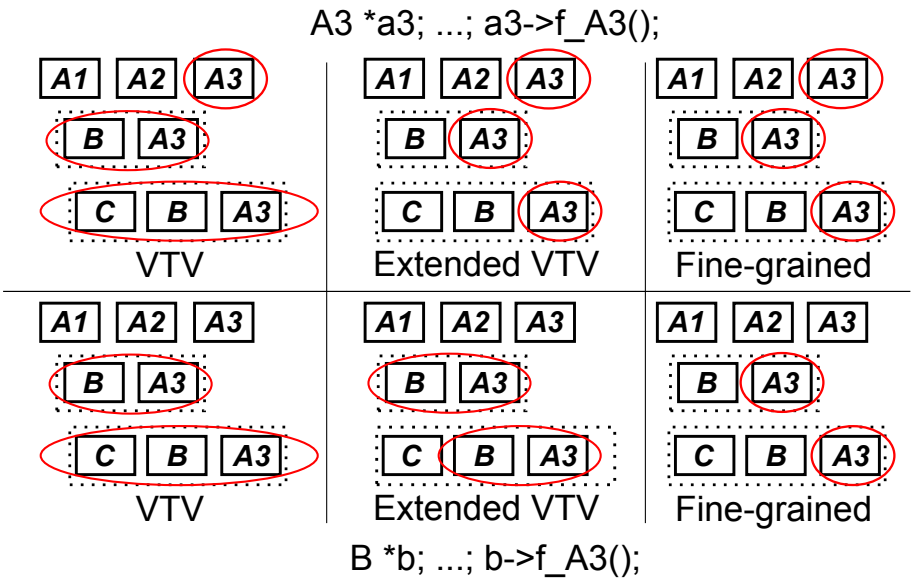


Figure 4.4: Example of VTable sets for two particular call-sites. The class hierarchy is reused from Figure 4.1. Each class defines a function *f_className*, while it does not overload any of its parents. The red circles are used to highlight the VTables added to the set accessible at the particular call-site for each protection scheme. The top row shows the VTables sets associated with a call-site of type *A3* using method *f_A3*. In case of the original VTV all VTables of all descendants of *A3* are added to the set, including VTables inherited from *A1* and *A2*. The extended VTV ensures that only the VTables generated due to *A3* are part of the set. For this call-site there is no difference between the extended and the fine-grained versions. The second call-site involves the type *B*, but the same method *f_A3*. The original VTV shows the same problem as before, but in this instance even the extended version is not optimal. Since the call-site cannot differentiate between the two VTables of *B*, both of them and their descendants need to be added to the set. By using the fine-grained approach, we identify that the call-site leverages the second VTable of *B* specifically. Thus we only add this particular VTable its descendant to the set.

Besides being an optimal VTable protection policy, fine-grained verification also has the advantage of offering guaranteed protection for VTable-based call-sites. In the case of VTV, it is still possible to call functions unrelated to the current call-sites, or to use VTable offsets that overflow the VTable in use (as highlighted in Section 4.5.2). Fine-grained protection guarantees by design, that the selected offset is always valid within all accessible VTables (since the layouts match). Furthermore, the function pointer at the offset always refers to the method specified in the source code or one of its overloaded variants. This means that attackers are unable to take advantage of mismatches in function prototypes or argument usage to further corrupt the program flow. All potential targets are also theoretically valid at the particular call-site, thus it is the responsibility of the programmer to design the methods with compatible semantics. While this does not stop all exploitation attempts against the program, it does eliminate the use of VTable-based call-sites as control-flow hi-

jacking targets. Since these represent 90% of all indirect call-sites in modern C++ programs [127] the vulnerability surface is reduced significantly, while maintaining overhead low enough to be acceptable by software development companies. We believe that other vulnerability vectors should also be protected using similar defenses focusing on the underlying semantics, instead of generic, coarse-grained protection mechanisms.

4.5 Evaluation

4.5.1 Microbenchmark evaluating correctness

The complexities of VTable inheritance policies within C++ make it difficult to guarantee a correct implementation using only a simple intuitive design. Thus we propose using a custom-designed microbenchmark to prove correctness in both the proposed and future VTable protection mechanisms. The point of the microbenchmark is to cover all inheritance scenarios as well as call-sites to ensure that the VTable protection implementation does not break valid C++ semantics, while also not including unneeded VTables in the corresponding sets.

We design the microbenchmark to cover all valid class hierarchies, including combinations of multiple and virtual inheritance. Since class hierarchies can have infinite size in theory and the possible number of class inheritance combinations increases exponentially, we define some practical limits to the class hierarchies we generate. The first limit is the maximum number of classes included in the hierarchy. The second option is the maximum number of base classes. Given these limits we generate every possible class hierarchy and include it in a source file of its own. These files make up the microbenchmark for evaluating the correctness and precision of VTable protection.

In order to trigger the use of all potential VTables, we create objects corresponding to each class and cast these objects to all possible valid dynamic casting targets. Finally we introduce VTable based call-sites for each of the cast results. The benchmark is evaluated along 3 axes: (i) precision of call-site type inference, (ii) correct execution with respect to C++ semantics, and (iii) optimality of VTable set content.

The call-site type inference is validated by statically analyzing the binary generated for each set. All VTable verification calls within a testing function corresponding to class `X` should only use VTable sets associated with `X`. This is defined by the program logic, which specifies that the argument points to a valid object of type `X`. Any outliers to this rule raises an error in the benchmark to signal that type inference is overly conservative. Correct execution is checked by running all binaries and monitoring that VTable verification does not raise errors for any combination of run-time object and call-site.

While previous papers considered enough the binaries to execute successfully, with ShrinkWrap we aim for optimal VTable sets. Thus we also evaluate if none of the VTable sets include unused entries. By construction, the microbenchmark

covers all combinations of objects and call-sites that may occur within valid and semantically correct C++ code. This property of completeness allows us to identify unused entries in the VTable sets correctly and to report them. A VTable protection is only considered correct and optimal if it is capable of running the benchmark successfully, while reporting no unused entries in its VTable sets.

The existing implementation of VTV [127] in GCC 4.9.2 fails the microbenchmark both in terms of call-site type inference, as well as the optimality of the VTable sets. Our proposed redesign of call-site type inference fixes the first problem, but only fine-grained VTable set generation is capable of passing all three aspects of our benchmark. This means that our proposed solution not only supports full C++ semantics, but it is optimal in terms of the VTable set contents. We found the benchmark incredibly valuable during the implementation process to identify corner-cases within the C++ standard. We have observed several instances, where Google Chrome would execute correctly with a particular implementation variant, while the benchmark would fail. As a result of our experiences, we highly recommend that all future research on VTable protection leverages this benchmark or a similar one. This will ensure that theoretical solutions are backed up by a correct implementation making the solutions sound in face of a determined attacker. To facilitate this aspect, we will make our micro-benchmark public as open-source code.

4.5.2 Chrome

Building Chrome with VTable protection

While building Chrome with precise call-site type inference, we noticed that VTable verification fails in a single place within the Skia library for the Chrome version we wanted to test. The particular fragment of code was eliminated in a later version, using the patch with identifier *76f5cc6e9e87ff247c3ef1f4b3fb03668db06e2e*, as it was deemed unnecessary and restricting the class hierarchy. For the evaluation we changed two lines of code to eliminate the artifact without affecting run-time behavior. The code itself is a classic case of benign type confusion, where a base class is statically casted into one of it's sub-classes, when it truly is just an object of the base class. The code does not crash with current compilers as the layout of the objects stays identical from a VTable perspective, since the sub-class does not implement any new methods. The cast operation is still invalid considering high-level C++ semantics and should be avoided in clean code.

We compiled the Chrome browser (version 42.0.2305.0, 64 bit) without VTable protection, with the original VTV and with the proposed enhancements deployed in multiple stages. This resulted in the following variants used throughout the evaluation:

- Original: VTV as it is implemented in GCC 4.9.2.
- Call-site: Applying precise call-site type inference to "Original" (described in 4.3.1).

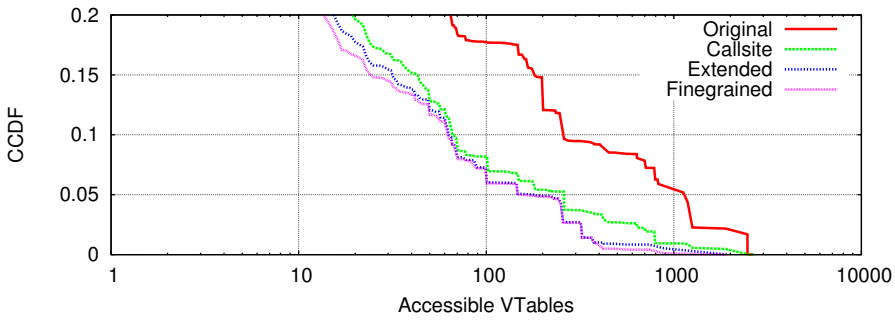


Figure 4.5: Complementary cumulative distribution function between the number of call-sites and the number of VTables they allow. The X-axis represents the number of VTables allowed at a call-site. The Y-axis represents the number of call-sites that use more than the given number of VTables.

- Extended: The proposed extension to VTV filtering on top of "Call-site" (described in 4.4.1).
- Fine-grained: The proposed fine-grained filtering scheme on top of "Call-site" (described in 4.4.2).

We plan to release all of these variants as patches to GCC 4.9.2, which will hopefully help developers to swiftly integrate the changes into the main development tree. Having the code as open-source will also allow other researchers to scrutinize our work in the future.

Security evaluation

The security evaluation of the original VTV and the different proposed enhancements is performed along two axes. The first is the size of the VTable sets allowed at each of the virtual call-sites within the program. The lower this number is, the smaller the control an attacker can exhibit when corrupting a VTable pointer. This simple metric for both source- and binary-based solutions, allows for easy comparisons against existing and future systems. For example, Prakash et. al. [9] also use this metric to evaluate their solution, vfGuard. The main issue with this metric is the lack of a proper baseline, since C++ semantics require that virtual call-sites allow a set of VTables based on the class hierarchy. To the best of our knowledge, ShrinkWrap is the first work to extract these sets precisely from the source code.

In an attempt to provide an alternative baseline, we propose a second metric focused on functional semantics instead of raw VTables. At each virtual call-site we analyze the set of methods accessible via the set of VTables allowed within the protection scheme. Instead of counting the addresses, we count the number of unique method names that we encounter at each virtual call-site. Destructors are all counted as a single method name. A method name encompasses a group of polymorphic methods with compatible semantics. The intuition is that within the source every

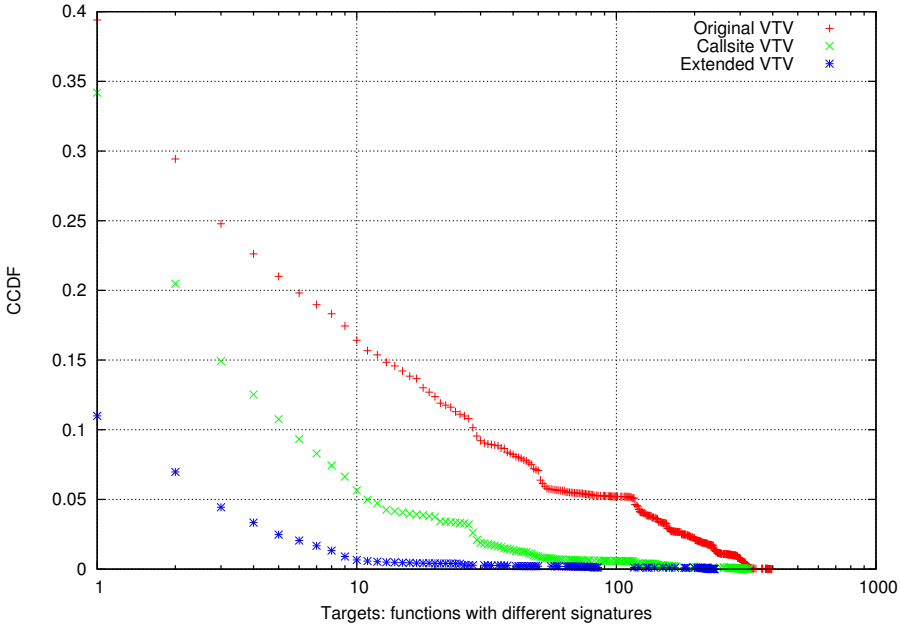


Figure 4.6: Complementary cumulative distribution function between the number of call-sites and the number of methods (by name) they can target. The X-axis represents the number of methods (by name) that a call-site can target. The Y-axis represents the number of call-sites that use more than the given number of methods (by name).

virtual call-site is specific to a given method name as specified by the developer. Precise VTable protection scheme is expected to enforce this property, otherwise an attacker can gain significant leverage, by diverting the control-flow to an unexpected method body, different from the one specified in the source code. This mechanism has been used [54; 55] for establishing the initial control over the return address, thus we consider it a serious threat to security.

The analysis corresponding to these two metrics are depicted in Figure 4.5 and Figure 4.6. One key observation is that around 2.5% of call-sites allow access to > 2500 VTables ($> 7.5\%$ of all VTables in Chrome). While the percentages are not impressive at first sight, one has to take into account the size of the Chrome binary (without VTV) used for these evaluations being 115MB in size after symbols are removed. This number suggests that an attack similar to the one presented by Schuster et. al. [113] might still be viable in the presence of VTV. The latter showed that main-loop gadgets were found in binaries as small as 1MB in size, which is smaller than 2.5% of the Chrome binary that we observe as being highly dangerous. The paper also claims that the other gadgets were successfully found within binaries smaller than 20MB. In the case of VTV, 7.5% of Chrome corresponds to somewhat

```
SetRemoteSSRCType(int,webrtc::StreamType,unsigned int)
SetFECStatus(int,bool,unsigned char,unsigned char)
RegisterObserver(int,webrtc::ViECaptureObserver&)
StartRender(int)
DeregisterEncoderObserver(int)
LastError()
ReceivedBWEPacket(int,long,unsigned long,webrtc::RTPHeaderconst&)
```

Figure 4.7: List of methods that an attacker can target at a particular call-site within Chrome. Even with the Original VTV enabled, the attacker can redirect execution to any of these 7 families of methods.

less code, but still within the same magnitude. These results suggest that VTV can be susceptible to the attack or a future variant of it.

Another point of concern is the large number of polymorphic method families accessible at each call-site, when using the existing implementation of VTV. More than 17% of call-sites allows attackers to target at least 10 different method families. In Figure 4.7 we present a concrete example to show the wide range of semantics accessible to an attacker at one such call-site. Some call-sites go as far as to allow access to more than 300 different method families, which is a significant attack surface, with potential to be exploited by resourceful adversaries. Figure 4.8 highlights an example of a class hierarchy, inspired by complex C++ projects, where an attacker can change the call-site's semantics for performing undesirable functionalities while original VTV is in place.

Precise call-site type inference has a significant impact on both metric, reducing the average number of VTables and methods accessible at call-sites, however it still suffers from corner-cases that could potentially become vulnerable in the future, e.g. around 6% of call-sites still allow access to 10 different method families or more. The Extended variant comes close to achieving the desired strictness according the method count metric, but it comes short in a small set of corner-cases. Fine-grained VTV is not shown in Figure 4.6 as all call-sites restrict access to a single method family, as desired when deploying strict VTable protection. The average VTable set size is also reduced to 27 compared to the average of 146 with the original VTV. This result is key to point out for binary-based VTable protection schemes that currently compare against VTV as a reference solution (such as [9]).

Performance overhead

Besides the security evaluation, it is also important to ensure that the proposed enhancements do not affect the low overhead offered by the original VTV implementation [127]. We evaluate the overhead imposed by the different variants using the Chrome browser. Our testbed is an HP Z230 i7-4770 3.4 GHz machine running Ubuntu Linux 12.04.5 with ASLR and turbo mode off to reduce the possible noise during the measurements. We compiled the Chrome browser (version 42.0.2305.0,

```
// Compile with GCC 4.9.2 VTV enable
// g++ -fvtable-verify=std -mpreferred-stack-boundary=2 -m32
#include <unistd.h>
#include <stdlib.h>
struct RefCounted {
        virtual void addRef() {}
        virtual void delRef() {} };
struct Logged {
        virtual void log() {} };
struct ProcessWrapper : virtual Logged, virtual RefCounted {
        virtual void run(char *path) {
            execlp(path, path, NULL);
        } };
int main(int argc, char ** argv) {
    // --Original Object Pointer--
    RefCounted *ptr = new RefCounted();
    // --Memory Corruption--
    ptr = (RefCounted*)(void*)(new ProcessWrapper());
    __asm__("push %0\n"::"r"("ls"));
    // --Hijacked Call-Site--
    ptr->delRef(); }
```

Figure 4.8: Proof-of-concept attack against VTV. An attacker corrupts an object pointer on the stack, changing its target to a subclass. The call-site for *delRef* will use the wrong VTable from within *ProcessWrapper*, since the appropriate up-casting is missing during the corruption. The call-site will redirect to the *run* method, taking the last stack entry as its argument (the string *"ls"* set up by the attacker).

64 bit) with the default release configuration as well as with the different variants of VTable protection added to the configuration. We evaluated performance across a series of popular browser benchmarks. Every benchmark is executed 10 times and with the average being taken as the final value. The results are depicted in Figure 4.9 and, as expected, the fine-grained VTV, the scheme we propose in this paper, performs better than the other variant of VTV. This mainly stems from the fact that the available targets at virtual call-sites are reduced the most in fine-grained VTV. Therefore, we stress that our proposal does not sacrifice performance for better security, but, instead, it is better in both performance and security, compared to the original VTV.

4.6 Related Work

Protecting return addresses stored on the stack [33] and support of non-executable data by many hardware processors and operating systems have raised the bar in software exploitation. Attackers have no way to inject code anymore, and they have to reuse existing code [116] by combining multiple bugs (one for taking control and one for leaking the process layout [121] for overcoming randomization [103]). Sophisticated exploits appeared and drove the community to seek a generic principle that could eventually offer sound protection for software, namely Control-Flow Integrity (CFI) [3].

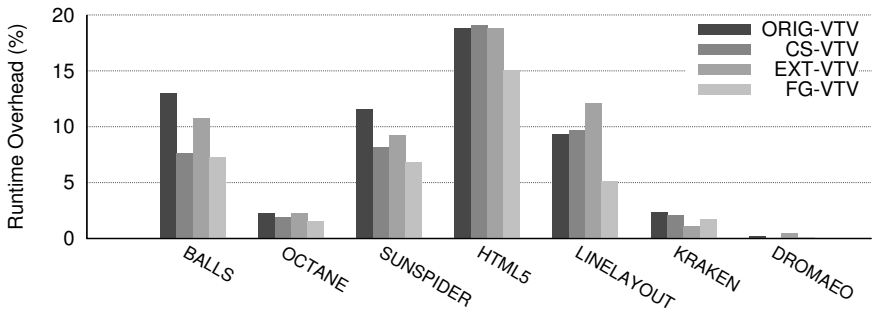


Figure 4.9: Performance evaluation of the proposed scheme. The overhead imposed by the proposed fine-grained VTable protection compared to the original VTV.

CFI suggests that binaries should be able to exercise only the control-flows that are allowed by the program's source. All indirect branches that happen at run-time should not be arbitrarily influenced by user input. Applying CFI in real-world software is challenging, since legacy applications cannot be recompiled, and even in the case where source is available, discovery of the complete control-flow graph is challenging [99], while the performance of validating call targets can produce overheads. Therefore, relaxed implementations [139; 141] were proposed that could be applied directly in binaries, but as it was quickly demonstrated, these implementations sacrificed security and therefore they are potentially exploitable [54; 38; 22].

Another approach for enforcing CFI is by leveraging certain hardware features, such as the Last Branch Record (LBR) debug registers, for performing anomaly detection in the last indirect branches a process has followed [102; 27]. Unfortunately it seems that an exploit can evade detection by inserting legitimate-looking gadgets in its ROP chain [55; 38; 22].

A particular set of CFI solutions focus on protecting only VTable pointers. The details of different variants are discussed in Section 4.2.2.

Last but not least, researchers have proposed methods for defending against use-after-free vulnerabilities based on custom allocators [5], which do not allow a memory area to host different type of objects during the life cycle of the process, or patching the developer's code for keeping track of dangling pointers [137]. These techniques protect *only* against use-after-free bugs, and they experience serious memory and computational overheads.

4.7 Conclusion

In this paper we revisited VTable protections. Although it was recently demonstrated that binary-based solutions that aim at protecting VTables fail to reconstruct

C++ semantics, and thus they are potentially vulnerable, we further argued that even when source code is available, the analysis is not straight-forward. We went through the state-of-the-art industry standard implementations, VTV, and highlighted weaknesses. Based on that, we formally modeled and designed an optimal solution for protecting VTables, and we implemented our proposal in GCC. In addition, we developed a testing methodology that can demonstrate that our analysis is correct, while it can also assist in evaluating similar VTable-protection frameworks. This paper suggests a formal guideline and evaluation framework for any methodology that aims at hardening binaries by protecting VTables.

4.8 Acknowledgment

This work is supported by the European Research Council through project ERC-2010-StG 259108-ROSETTA, by the Microsoft Research PhD Scholarship Programme through the project MRL 2011-049.9, by the Netherlands Organisation for Scientific Research through grant NWO 639.023.309 VICI “Dowsing”, and by the European Commission through the project SHARCS under Grant Agreement No. 644571.

5

METALLOC: Efficient and Comprehensive Metadata Management for Software Security Hardening

Abstract

Many systems software security hardening solutions rely on the ability to look up metadata for individual memory objects during the execution, but state-of-the-art metadata management schemes incur significant lookup-time or allocation-time overheads and are unable to handle different memory objects (i.e., stack, heap, and global) in a comprehensive and uniform manner.

We present METALLOC, a new memory metadata management scheme which addresses all the key limitations of existing solutions. Our design relies on a compact memory shadowing scheme empowered by an alignment-based object allocation strategy. METALLOC's allocation strategy ensures that all the memory objects within a page share the same alignment class and each object is always allocated to use the largest alignment class possible. This strategy provides a fast memory-to-metadata mapping, while minimizing metadata size and reducing memory fragmentation. We implemented and evaluated METALLOC on Linux and show that METALLOC (de)allocations incur just 3.6% run-time performance overhead, paving the way for practical software security hardening in real-world deployment scenarios.

5.1 Introduction

Many common software security hardening solutions need to maintain and look up memory metadata at runtime. Examples include bounds information to validate array references [6; 7], type information to validate cast operations [84], solutions that prevent use-after-free exploits [83; 137], and object pointer information to perform garbage collection [107]. While newer programming languages can often track such metadata in-band using fat pointers, previous efforts to implement in-band metadata management in systems programming languages, such as C or C++, have found limited applicability due to poor ABI compatibility and nontrivial overhead [41].

The alternative solution is to associate metadata information with the memory objects themselves, assuming we have a mechanism to map pointers to the appropriate metadata. Such a primitive is the key to implementing modern metadata management schemes, but it is also challenging because it needs to support all the possible memory objects (heap objects allocated with `malloc()`, globals, and stack objects) as well as minimize the performance and memory impact of metadata update and lookup operations.

Minimizing the impact of update operations is challenging, because allocation and deallocation of memory objects and their metadata occurs frequently during execution. Minimizing the impact of lookups is also challenging, since metadata lookup must be able to support interior pointers into nested classes, structures, and arrays—dictating support for range queries and disqualifying the use of space- and time-efficient hash tables.

Current state-of-the-art software hardening projects all rely on tailored, mostly one-off solutions for metadata management, but none of them simultaneously achieves low lookup and update impact in all cases. As a result, none of them provides a generic solution. Common approaches include tree-based metadata handling and memory shadowing.

Tree-based approaches [61; 83] store an interval node for each allocated object according to its bounds. Unfortunately, tree lookups can result in a prohibitive performance hit, as the tree depth is frequently in the double digit range (more than 1,024 memory objects). The lookup time is also unpredictable, as it varies with the object count. As a result, tree-based systems are unsuitable for most production situations.

Traditional memory shadowing, in turn, relies on a fixed pointer-to-metadata mapping [6; 7; 137]. The key design choice for this approach is the *metadata compression ratio*. The metadata compression ratio represents the number of metadata bytes that need to be tracked for each data byte. For example, assume we store one byte of metadata for each block of eight bytes. In this case the compression ratio is $\frac{1}{8}$. If we have a pointer p and an array of metadata starting at address q, we can compute a pointer to the metadata for object as $\frac{1}{8}p + q$. This way, metadata can be located very efficiently. However, choosing the appropriate compression ratio is difficult, as it enforces a minimum alignment on every memory allocation. Small

compression ratios result in inflated metadata size and a large tracking overhead, while large compression ratios result in significant memory fragmentation. In practice, memory management systems typically only guarantee alignment up to 8 or 16 bytes. This means that to keep the compression ratio reasonably small only a single byte of metadata is supported [6; 7]. Even then, this approach introduces prohibitive initialization time and memory overhead for large objects in case multi-byte metadata is needed. Finally, recent approaches rely on custom allocators to reduce the impact of memory shadowing on the heap, but cannot support efficient and comprehensive metadata management including more performance-sensitive objects on the stack [84].

In this paper, we propose METALLOC, a new metadata memory management scheme based on an efficient and comprehensive variable memory shadowing strategy. Our strategy builds on recent developments in heap [49] and stack [80] organizations to implement a variable and uniform pointer-to-shadow mapping and significantly reduce the performance and memory impact of metadata management. Our results show that METALLOC is practical and can support efficient whole-memory metadata management for several software security hardening solutions.

Summarizing, we make the following contributions:

- We propose a new memory metadata management scheme that supports interior pointers and is time- and space-efficient in both lookups and updates across all memory object types.
- We present a prototype implementation termed METALLOC, which demonstrates that efficient and comprehensive metadata management is feasible and widely applicable in practice.
- We present an empirical evaluation showing that METALLOC incurs a runtime performance overhead of just 3.6% for (de)allocations on SPEC2006.

5.2 METALLOC

As we have seen, one of the major limitations of state-of-the-art memory shadowing approaches is the difficulty of getting the compression ratio right. Because the right value may differ from application to application, the intuitive solution is to enable a variable compression ratio. This eliminates the fixed memory overhead associated with metadata shadowing and greatly reduces the allocation-time performance hit.

METALLOC's key goal is to implement a metadata management scheme handling all memory objects in a uniform and highly efficient manner, regardless of their allocation type (heap, stack, or global memory). There are two requirements to accomplish this goal. The first is to support a simple, efficient, and uniform mechanism to associate pointers with the compression ratio. The second is the ability to optimize the compression ratio as much as possible, ideally such that only a single metadata entry is needed for each object. METALLOC meets both these require-

ments by ensuring that all the memory objects within a memory page share a non-trivial common alignment, which is fixed as long as there are active objects within the page. This requirement serves as a basis for our scheme and is met by drawing from modern heap [49] and stack [80] organizations widely used in production, as discussed in Section 5.2.4.

Alignment relates directly to the compression ratio, namely an n-byte object alignment allows one *metadata entry* to be associated to every group of n bytes within said object. Having uniform alignment within each memory page allows METALLOC to associate compression ratios to the individual memory pages and to look them up using a mechanism similar to *page tables*. Such page tables also include the location of the metadata region corresponding to the individual pages. This mechanism is described in the following section.

5.2.1 Efficient retrieval of page information

Because lookups are expected to be very frequent, the page table design is very performance-sensitive. For this reason, METALLOC opts for a single-level page table design, which requires only one memory read for each lookup. We refer to this data structure as the *meta-page table*. Figure 5.1 shows the data structures of METALLOC, including the use of the meta-page table. Given a pointer, we split it in a page index and an offset. The page index is used as an index into the meta-page table, which is an array stored at the page table base. This page table base is a compile-time constant and therefore requires no extra memory read. The entries in this array are eight bytes large, split between the seven-byte *metabase* pointer and the one-byte base-2 logarithm of the memory alignment. The metabase points to the start of an array of metadata entries available for this page table entry. The size of the entries of the metadata array is determined by the *metasize* compile-time constant, which specifies the amount of metadata per object. It should be noted that objects larger than the alignment size have multiple metadata entries, each with the same contents. The offset part of the original pointer divided by the alignment serves as an index into the metadata array, allowing the correct metadata entry to be located for the specified object.

While the size of the meta-page table (for x86-64 systems using 48-bit virtual addresses and 4096-byte pages) is theoretically 2^{36} entries or 512 GB, only pages corresponding to address ranges currently in use by the process need to be allocated. To implement this strategy, METALLOC reserves the required virtual memory area in advance and relies on demand paging to lazily link the required page table pages to physical memory.

5.2.2 Static versus dynamic metadata

The scheme presented so far assumes a fixed-sized metadata entry which is statically initialized at the start of the object's lifetime. For objects whose size exceeds their

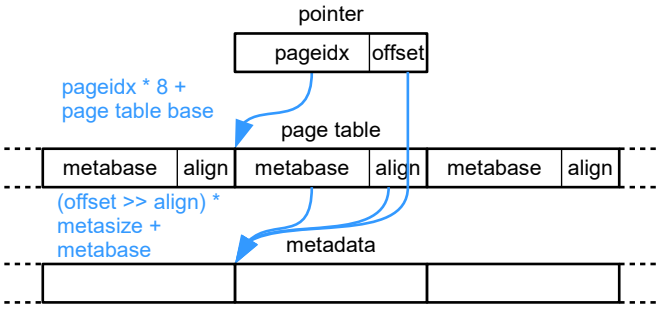


Figure 5.1: METALLOC's data structures

alignment there are multiple copies of this metadata entry in memory. A metadata entry can be an integer, a small struct or even a pointer, which can refer to further metadata of arbitrary dynamic size.

In the simplest possible setup, each metadata entry is filled with a compile-time constant, such as a type index. This scheme allows the instrumentation to identify certain predetermined object characteristics at run time, with the lowest possible overhead. In this use case, initializing the metadata is cheap, as it involves a simple `memset` operation for the compressed metadata range corresponding to the object. Given the use of variable compression rates, the amount of metadata that needs to be initialized is minimal.

Alternatively, the metadata can include a pointer to information created at compile time in global memory. This scheme may be, for example, used to support in-depth type tracking when implementing precise garbage collection for languages such as C/C++ [107]. From an overhead perspective, this is similar to the constant metadata scenario, as the pointer itself is just another numeric constant, whose value is fixed at compile time.

Finally, the metadata pointer might be generated at run time. In this case, a *metadata node* is allocated whenever a new memory object is allocated and deallocated when it is freed. This scheme is relevant to instrumentation tailored to individual object instances, rather than broad object groups. Sharing lifetimes is trivial on the stack and in global memory, but leads to interesting design decisions when dealing with the heap. Intuitively, the fastest solution is to increase the allocated size of the object itself and store the metadata node in-band. In practice, we found this solution to result in significant performance degradation due to its negative caching impact. The alternative is to perform a second heap allocation dedicated to the metadata node itself. While we expected a nontrivial performance hit for applications where allocations dominate the run time, we found the impact to be reasonable in practice. In addition, most applications do not require object-specific metadata tracking and can safely refrain from using dynamic metadata, as it incurs the highest run-time and memory overhead.

5.2.3 Instrumentation across memory types

Globals. Global memory is the simplest to deal with from a memory shadowing perspective. Global allocations only occur at load time and the amount of global data is typically limited. These properties suggest that the default 8-byte alignment within the system works reasonably well for this allocation type, thus we can just associate every global data memory page with this alignment in the meta-page table. We simply instrument binary files (executables and dynamic libraries) to allocate metadata pages and set up the meta-page table entries corresponding to their data sections as soon as the binaries are loaded.

Stack. Stack pages share the property of long lifetimes with global memory (associated with the stack for the lifetime of the thread), but they are unique in that object allocations within the pages occur frequently and with minimal run-time cost. A simple solution is to restrict stack objects to the conservative 8-byte alignment [6], but this makes tracking multi-byte metadata prohibitively expensive. METALLOC solves this challenge by leveraging recent advances in shadow stack solutions (recently integrated in mainstream compilers) [80], which split the program stack into a primary and a secondary stack. METALLOC uses a multi-stack approach with a primary and a number of secondary stacks. The primary stack preserves all the stack objects not subject to the current instrumentation, including ABI specific elements such as return addresses and arguments, while the secondary stacks store the remainder. The secondary stacks each are designed for a particular class of object sizes with appropriate non-trivial alignment to improve the compression ratio for metadata tracking. This design ensures that METALLOC only needs to care about the secondary stacks, which are free from ABI specific restrictions. We propose using the same heuristic for all instrumentations, namely moving all objects which can potentially be subject to memory corruption attacks (either address taken or address used in unpredictable manner) to one of the secondary stacks. In practice the secondary stacks end up composed mostly of arrays and some address taken integers. As a result, we propose enforcing a relatively large alignment on each object within every secondary stacks. While some memory fragmentation does occur with address taken integers, it only affects a small portion of the entire memory address space of the program and it will not break program functionality in general. Our current design uses one secondary stack for small and medium objects having a fixed alignment of 64 bytes, and another secondary stack for large objects where 4096 byte alignment is enforced. METALLOC instruments the allocation of program stacks to create the secondary stacks and to also allocate the corresponding metadata pages and to set up the appropriate meta-page table entries.

Heap. In order to meet our requirements, METALLOC uses a heap allocator designed around the concept of object sizes. Instead of keeping track of the heap as a whole, it operates with size-specific free-lists instead. Whenever a free-list becomes empty, the allocator can request new memory pages from the system, which are then associated with this particular free-list until they are released back to the system.

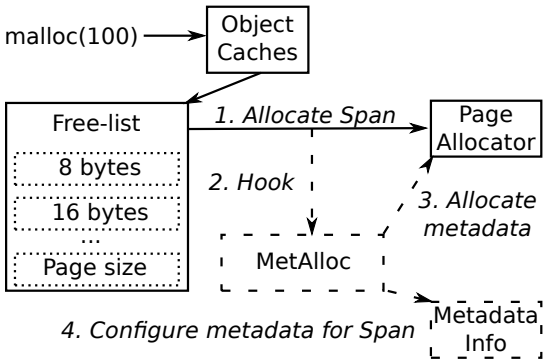


Figure 5.2: Heap metadata management using METALLOC

The allocator would then enforce the largest possible alignment for objects within each free-list without triggering too much fragmentation. Applying METALLOC to such a heap allocator is trivial as one just needs to monitor page request by the free-lists to associate the appropriate metadata information with the pages. Figure 5.2 summarizes this operation with the addition of potential object caches and a page allocator to improve performance.

5.2.4 Implementation specifics

While it is possible to build a new custom heap allocator which respects our design specification, we decided to build upon a proven, state-of-the-art allocator instead. We expect that for most complex C++ applications the heap allocator has a significant impact on performance, thus a proven allocator is key. The tcmalloc allocator (developed and used by Google) features a memory organization which matches our requirements. Neither tcmalloc nor our modifications affect ABI compatibility, so no changes are needed in the operating system and external libraries do not need to be recompiled.

For the stack we decided to extend the implementation of SafeStack [80], as it is advertised as a replacement for stack canaries going forward and it is becoming a core component within the LLVM project. Its interpretation of safe and unsafe objects also matches up well with our definition of objects requiring instrumentation. SafeStack does not affect ABI compatibility.

5.3 Applications

Efficient metadata tracking enables a wide range of valuable instrumentation tools to be used with production systems where performance overhead is a key characteristic. In the following we present a couple of key examples of such instrumentation. This list is by no means exhaustive and we hope that readers find other innovative uses

of the framework. In this (short) paper, the applications serve as motivation for our work and we will focus our evaluation on the framework itself.

5.3.1 Write Integrity Protection

Recent developments in attack techniques [23; 44; 113] show the need to enforce additional data integrity within the program besides the classic control-flow integrity. Both Microsoft's WIT [6] and Oracle Application Data Integrity have looked into the topic of restricting the target addresses of memory writes using a coloring mechanism. In these schemes each memory location is associated with a given color and the instrumentation alongside each memory write checks if the target location matches the color of the pointer/instruction. In the case of both of these systems, the color for the memory location is tracked using metadata shadowing with a fixed compression ratio. Replacing these systems with METALLOC can lead to substantial improvements in allocation performance.

5.3.2 Bounds Checking

Efficient bounds checking has been proposed in the past to counter buffer overflow vulnerabilities, but none of the solutions ended up in production systems due to the performance and memory overheads they bring. One particularly efficient example is Baggy Bounds Checking [7] which offers a strong protection model with limited memory overhead, based on fixed compression metadata. Its primary deficiency is the need to allocate objects in slots with sizes in the powers of two, a requirement that is typically not enforced in generic heap allocators due to the potential for high internal fragmentation. The system can be rebuilt without the alignment requirements but that would require tracking base pointer and size information for every object, which leads to performance and memory issues with the fixed compression ratio (it is prohibitive to store 16 bytes of metadata for every 8 data bytes). METALLOC and its variable compression ratio can help to deal with the problematic large object allocations, ensuring consistently low overhead across applications even when using multiple metadata bytes.

An alternative implementation of bounds checking is Light-weight Bounds Checking [67]. This system detects out-of-bounds accesses at the memory access time instead of during the pointer arithmetic. The system injects guard zones between objects and fills them with a random byte value to detect any access into these regions. A memory access is safe if it returns a different value, but real data might also accidentally match the guard value. An additional check is performed in the latter case to filter out false positives, but on average it is only performed with probability 1 in 256. This check retrieves a metadata bit associated with the address which specifies if it belongs to real data or one of the guard zones. Light-weight Bounds Checking uses a fixed compression ratio shadowing scheme of one metadata bit for every byte of data in the program. However, metadata retrieval is avoided on the fast-path of

this scheme with little impact on performance. As a result replacing the existing metadata tracking with METALLOC only yields benefits to the system. The existing system uses a hierarchical metadata storage system requiring two memory accesses to retrieve the metadata bit. METALLOC also performs two memory accesses, but it involves more pointer arithmetic instructions. It is safe to say that the fast-path behavior will easily hide the small difference in retrieval overhead. On the other hand the variable compression ratio of METALLOC reduces allocation overhead, which can be significant in many applications. As a result, Light-weight Bounds Checking can also benefit from using METALLOC for its metadata tracking.

5.3.3 Type Confusion Detection

Recently type confusion vulnerabilities received significant attention as an alternative memory corruption mechanism which is not covered well by static analysis and run-time checkers. CaVer [84] was designed as an efficient system to dynamically track type information and to perform type validation at potentially vulnerable cast locations. It uses the metadata system tracking in LLVM for heap objects, but reverts to red-black trees for stack and global allocations. As such, it requires additional operations during metadata retrieval to identify the type of the pointer. By using METALLOC, CaVer gains access to uniform pointer handling and low overhead irrelevant of the memory usage pattern. This is especially beneficial when considering the excessive overhead reported in CaVer for Firefox, which was attributed to its use of stack variables.

5.3.4 Dangling Pointer Detection

Use-after-free vulnerabilities represent the most prominent attack vectors in today's browser landscape [83]. While a lot of effort is invested to detect these vulnerabilities via static analysis and software testing, they typically manifest in highly specialized contexts, making them hard to detect and to fix preemptively. As such, a couple of systems have been suggested recently to mitigate the underlying reason for the vulnerabilities, dangling pointers [83; 137]. These systems rely on tracking heap allocations and their connectivity at run time. When an object is freed, the systems identify whether there are any pointers still pointing into the object being released. These pointers are then set to a benign value of NULL to mitigate potential memory dereferences using them. Systems for tracking dangling pointers share an underlying design based on three core data structures. The first is the object map, which identifies heap objects based on any pointer into the object itself (at any offset). This is equivalent to object metadata tracking. DangNull [83] uses red-black trees to track heap allocations, but as discussed in section 5.1, this scheme is susceptible to heavy and unpredictable overhead. FreeSentry [137] uses a label based system, which is equivalent to the fixed compression ratio metadata shadowing. This scheme offers fast fixed-time metadata retrieval, but incurs significant allocation-time and memory

overhead. In contrast, METALLOC combines low allocation- and lookup overhead with efficient memory usage.

5.4 Evaluation

To measure the performance impact of metadata tracking, we instrumented all the C and C++ SPEC2006 [68] benchmarks to observe the overhead it introduces. As a baseline, we compiled the applications with SafeStack enabled since it is advertised as a viable replacement for stack canaries, even showing lower overhead on some benchmarks [80]. We also use tcmalloc as the heap allocator for our baseline, as it can have very different run-time performance compared to the system allocator. This decision is also motivated by the fact that we observed a 10% improvement of execution times when using tcmalloc on SPEC2006 (geometric mean). This improvement increases to 17% when considering only the C++ benchmarks. For each benchmark, we used the median run time over 16 runs on a Xeon E5-2630 running CentOS Linux 7.2 64-bits.

Figure 5.3 shows the overhead introduced with the different configurations of METALLOC. We evaluated creating and initializing both 1 and 8-bytes of metadata for all objects. These setups correspond to different instrumentation types, like write integrity tracking or type hashes. The overhead numbers are very low, with the maximum being around 20% for perlbench and the geometric mean being 3.6% for one byte of metadata and 3.7% for eight bytes. The results also show that metadata size has a limited impact on the overall performance, showing that the variable compression ratio can help deal with applications requiring complex metadata. While the measured overhead only includes the metadata creation and initialization, not the instrumentation itself, the latter can be tuned with careful design and is the topic of future work using METALLOC.

5.5 Conclusion

In this paper, we presented METALLOC, a new memory metadata management scheme for software security hardening solutions. Our design is both comprehensive—given that it can handle whole-memory object metadata in a uniform and transparent way—and efficient—given that it yields a run-time performance overhead of just 3.6% in practice for (de)allocations. We believe METALLOC can bring many instrumentation solutions within reach for adoption in practice, allowing, for example, many vulnerability mitigation techniques to improve software security in an efficient and backward compatible fashion.

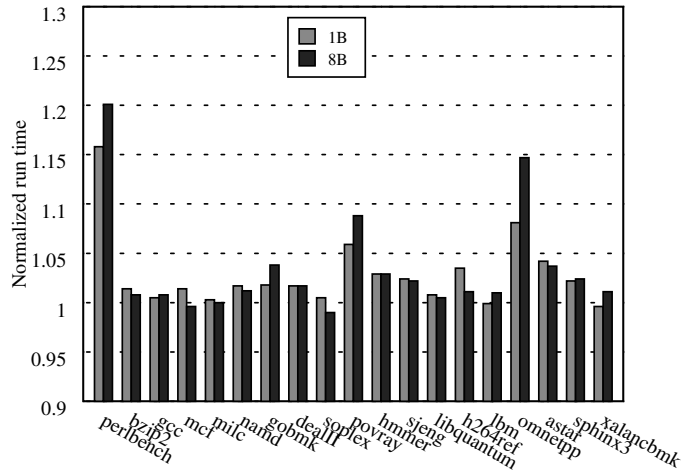


Figure 5.3: C/C++ SPEC2006 overhead with different configurations of METALLOC. 1(8)B represents the configuration with 1(8)-byte metadata entries.

5.6 Acknowledgment

This work is supported by the Netherlands Organisation for Scientific Research through grant NWO 639.023.309 VICI “Dowsing”, and by the European Commission through the project SHARCS under Grant Agreement No. 644571.

6

TypeSan: Practical Type Confusion Detection

Abstract

The low-level C++ programming language is ubiquitously used for its modularity and performance. Typecasting is a fundamental concept in C++ (and object-oriented programming in general) to convert a pointer from one object type into another. However, downcasting (converting a base class pointer to a derived class pointer) has critical security implications due to potentially different object memory layouts. Due to missing type safety in C++, a downcasted pointer can violate a programmer's intended pointer semantics, allowing an attacker to corrupt the underlying memory in a type-unsafe fashion. This vulnerability class is receiving increasing attention and is known as *type confusion* (or *bad-casting*). Several existing approaches detect different forms of type confusion, but these solutions are severely limited due to both high run-time performance overhead and low detection coverage.

This paper presents TypeSan, a practical type-confusion detector which provides both low run-time overhead and high detection coverage. Despite improving the coverage of state-of-the-art techniques, TypeSan significantly reduces the type-confusion detection overhead compared to other solutions. TypeSan relies on an efficient per-object metadata storage service based on a compact memory shadowing scheme. Our scheme treats all the memory objects (i.e., globals, stack, heap) uniformly to eliminate extra checks on the fast path and relies on a variable compression ratio to minimize run-time performance and memory overhead. Our experimental results confirm that TypeSan is practical, even when explicitly checking almost all the relevant typecasts in a given C++ program. Compared to the state of the art, TypeSan yields orders of magnitude higher coverage at 4–10 times lower performance overhead on SPEC and 2 times on Firefox. As a result, our solution offers superior protection and is suitable for deployment in production software. Moreover, our highly efficient metadata storage back-end is potentially useful for other defenses that require memory object tracking.

6.1 Introduction

Type confusion bugs are emerging as one of the most important attack vectors to compromise C++ applications. C++ is popular in large software projects that require both the modularity of object-oriented programming and the high efficiency offered by low-level access to memory and system intrinsics. Examples of large C++ programs are Google Chrome, large parts of Microsoft Windows and Firefox, and the Oracle Java Virtual Machine. Unfortunately, C++ enforces neither type nor memory safety. This lack of safety leads to type confusion vulnerabilities that can be abused to attack certain programs. Type confusion bugs are an interesting mix between lack of type safety and lack of memory safety. Generally, type confusion arises when the program interprets an object of one type as an object of a different type due to unsafe typecasting—leading to reinterpretation of memory areas in different contexts. For instance, it is not uncommon for a program to cast an instance of a parent class to a descendant class, even though this is not safe 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 of the descendant class that do not exist for the given object, it may use data, say, as a regular field in one context and as a virtual function table (vtable) pointer in another. Exploitable type confusion bugs have been found in a wide range of software products, such as Adobe Flash (CVE-2015-3077), Microsoft Internet Explorer (CVE-2015-6184), PHP (CVE-2016-3185), and Google Chrome (CVE-2013-0912). This paper shows how to detect type confusion with higher detection coverage and better performance than existing solutions.

TYPESAN: always-on type checking Current defenses against type confusion [105; 84] are impractical for production systems, because they are too slow, suffer from low coverage, and/or only support non-polymorphic classes. The greatest challenge in building an always-on type checker is the need for per-object metadata tracking which quickly becomes a bottleneck if the program allocates, frees, casts, and uses objects at high frequency (e.g., on the stack).

To address the high overhead and the low coverage of existing solutions, we present TYPESAN, an explicit type checking mechanism that uses LLVM-based instrumentation to enforce explicit type checks. Compared to previous work, TYPESAN provides extended coverage and massively reduced performance overhead. Our back-end uses a highly efficient metadata storage service (based on a shadowing scheme with a variable compression ratio) to look up types from pointers. This limits the amount of data written for large allocations (such as arrays of objects) while at the same time supporting efficient and scalable lookups, requiring only 3 memory reads to look up a type. We envision this new type of metadata storage to be also useful for other sanitizers, e.g., to verify memory safety, and we plan to explore further applications in future work.

We primarily envision TYPESAN as an always-on solution, making explicit type checks practical for commodity software. Used in attack prevention mode, TYPE-

Checker	xalancbmk	soplex	omnetpp	dealII
CaVer	24 thousand	0	0	0
TYPESAN	254 mln	209 thousand	2.0 bln	3.6 bln

Table 6.1: Coverage achieved by the type checkers on SPEC. The numbers represent the number of downcasts verified by each of these systems while executing the reference workloads. The CaVer numbers are from the original paper to ensure a fair comparison.

SAN-hardened binaries are shipped to end users and terminate the program on bad casts, thereby preventing zero-day type confusion exploits. Combined with liveness reports for modern software (like the Google Chrome and Mozilla Firefox crash reporters), such a deployment signals the developers about potentially missing type checks. In addition, TYPESAN can be used in software testing where TYPESAN identifies potential bad casting in the source code. In relaxed mode, TYPESAN simply logs all bad casts to scan for underlying vulnerabilities, e.g., when running a test suite.

We have implemented a prototype of TYPESAN for Linux on top of LLVM 3.9. Our prototype implementation is compatible with large source code bases. We have evaluated TYPESAN with the SPEC CPU2006 C++ programs and the Firefox browser. Compared to CaVer [84], the current state-of-the-art type confusion detector, we decrease overhead by a factor 3–6 for the SPEC benchmarks they reported on while simultaneously increasing the number of typecasts covered by checks by several orders of magnitude (see Table 6.1 and Table 6.2 for more details).

Contributions We make the following contributions:

- A design for high-performance, high-coverage typecast verification on legacy C++ code that is 3–6 times faster than the state-of-the-art detector with lower memory overhead and orders of magnitude more typecasts.
- A thorough evaluation that shows how our design delivers both nearly complete coverage and performance that is suitable for production usage.
- An automatically generated test suite for typecasting verification to ensure that all different combinations of C++ types are properly handled.
- An open-source implementation of our TYPESAN design, available at `https://github.com/vusec/typesan`.

6.2 Background

In this section, we first explain typecasting in C++ and how doing so incorrectly can lead to vulnerabilities. Afterwards, we discuss existing defenses against type confusion.

Checker	xalancbmk	soplex
CaVer	29.6%	20.0%
TypeSan	7.1%	1.8%

Table 6.2: Performance overhead achieved by the type checkers on SPEC. The CaVer numbers are taken from the original paper to ensure a fair comparison. The comparison only uses the applications CaVer has been evaluated on by its authors.

6.2.1 Type confusion

Object-oriented programming languages such as C++ allow object pointers to be converted from one type into another type, for example by treating an instance of a derived class as if it were an instance of one of its ancestor classes. Doing so allows code to be reused more easily and is valid because the data layout is such that an object from a derived class contains the fields of its parent classes at the same relative offsets from each other.

In our discussion on the safety of type conversions (or typecasts), we will use the following terminology: the *run-time type* refers to the type of the constructor used to create the object, the *source type* is the type of the pointer that is converted, and the *target type* is the type of the pointer after the type conversion. Since the program may treat objects as if they are instances of their ancestor types, an object pointer should always refer to an object with a run-time type that is either equal to or a descendant of the pointer type. Therefore, a type conversion is always permissible when the target type is an ancestor of the source type. A compiler can verify such casts statically, because if the source type is a descendant of the target type it implies that the run-time type is also a descendant. We refer to this type of conversion as an *upcast*.

If, on the other hand, the target type is a descendant of the source type, the conversion may or may not be permissible depending on whether the run-time type is either equal to or a descendant of the target type. This is impossible to verify in the general case at compile time because the run-time type is not known to the compiler, due to inter-procedural/inter-component data flows. We refer to this type of conversion as a *downcast*. Downcasts require run-time verification to ensure type safety. Incorrect downcasts may allow attackers to exploit differences in the memory layout or semantics of the fields between the target type and run-time type.

The C++ programming language permits both upcasts and downcasts and allows the programmer to specify whether downcasts should be checked at run time. Specifically, the language provides three fundamental types of casts: `reinterpret_cast`, `dynamic_cast`, and `static_cast`. Dynamic casts are enforced at run time with an explicit type check and are therefore a safe but expensive way to ensure type safety. Static casts on the other hand only verify whether the conversion could be a valid upcast or downcast based on the source and target types. This lack of an online check can easily lead to type confusion when the underlying type observed at run time differs from the expected type in the source code.

As an example, in V8Clipboard in Chrome 26.0.1410.64, we find the following static cast:

```
static_cast<HTMLImageElement*>(node)->cachedImage()
```

Here, the program explicitly casts an image `node` to an `HTMLImageElement` without properly checking that it is of the right type. Unfortunately, `node` could be an SVG image, which is of a sibling class and has a much smaller vtable than `HTMLImageElement`. Note that the program immediately calls `cachedImage()` on the invalid object which leads to a virtual function call that erroneously interprets the memory adjacent to the SVG image's vtable as code pointers.

If the program would check all static casts dynamically, we would not run into the type confusion problem (except for explicitly forced "problems" through reinterpreted casts). However, casting is such a common operation that the overhead of checking all static casts dynamically is significant and therefore, C++ allows the programmer to choose an explicit run-time cast only where "needed" (according to the programmer).

For completeness, we mention that the last form of casting, `reinterpret_cast`, forces a reinterpretation of a memory area into a different type. It allows a programmer to explicitly break underlying type assumptions.

6.2.2 Defenses against type confusion

In recent years, several projects have tried to address the type confusion problem. There are two main types of approaches: those based on vtable pointers embedded in the objects and those based on disjoint metadata. Solutions based on vtable pointers have the advantage that they do not need to track active objects but they have the fundamental limitation that they cannot support non-polymorphic classes, which do not have vtables, without breaking binary compatibility.

Examples of vtable-based solutions are UBSan [105] and Clang Control-Flow Integrity [29] (CFI). UBSan instruments static casts to execute an explicit run-time check, effectively turning them into dynamic casts. UBSan requires manual blacklisting to prevent failure on non-polymorphic classes. Unfortunately, the existing type check infrastructure that is available in C++ compilers is inherently slow (as it was designed under the assumption that only few dynamic checks would be executed with the majority of checks being static). This is another reason why type-safety solutions did not see wide adoption. Therefore, UBSan is intended as a testing tool and not as an always-on solution due to its prohibitive overhead. Clang CFI is designed to be faster but has not published performance numbers. Moreover, like all solutions in this group, it cannot support non-polymorphic classes.

CaVer [84], on the other hand, uses disjoint metadata to support non-polymorphic classes without blacklisting. Unfortunately, the overhead is still prohibitively high due to inefficient metadata tracking (especially on the stack) and slower checks, reaching up to 100% for some browser benchmarks. Because CaVer cannot rely

Checker	Poly	Non-poly	No blacklist	Tracking	Threads
UBSan	✓				✓
Clang CFI	✓		✓		✓
CaVer	✓	✓	✓	✓	limited
TYPESAN	✓	✓	✓	✓	✓

Table 6.3: High-level feature overview of checkers.

on vtables (present only in polymorphic objects), it must track all live objects. In particular, CaVer uses red-black trees to track stack-allocated objects and a direct mapping scheme based on a custom memory allocator for heap-based objects. As a consequence, it has to determine the correct memory region for each pointer to be checked and it cannot handle stack objects shared between threads even if proper synchronization is used in the application. In addition, as shown in Section 6.9.2, CaVer has poor object allocation coverage in practice, ultimately leading to reduced type-confusion detection coverage. For example, CaVer reports only 24k verified casts on xalancbmk and none on all other SPEC CPU2006 C++ benchmarks, while we show that four benchmarks actually have a large amount of relevant casts with their total numbers is in the billions. As such, TYPESAN is the first solution that provides efficient and comprehensive protection against type confusion attacks in the field, protecting users from vulnerabilities not found during testing.

In this paper we introduce TYPESAN, a generic solution for typecast verification based on object tracking, that supports all types of classes with no need for blacklisting. Moreover, we cover a very large percentage of all relevant casts at an acceptable overhead. Table 6.3 gives a high-level comparison of the typecast verification solutions presented here. UBSan and Clang CFI are restricted to polymorphic types, with UBSan requiring further blacklisting to handle certain code bases. CaVer and TYPESAN also support non-polymorphic types, but this comes at the cost of needing to track the type for each object. CaVer further comes with the limitation that threads cannot share stack objects safely even with proper synchronization. Table 6.3 and Table 6.4 show which allocation types are tracked in practice by tracking solutions. See Section 6.5.1 for a more in-depth discussion of the various allocation types. CaVer officially supports stack and global data, but missed such bad casts in our coverage tests (see Section 6.9.2).

6.3 Threat model

We assume that the attacker can exploit any type confusion vulnerability but is unable to perform arbitrary memory writes otherwise. Our type safety defense mechanism exclusively focuses on *type confusion*. Other types of vulnerabilities such as integer overflows or memory safety vulnerabilities are out of scope and we assume that orthogonal defense mechanisms protect against such vulnerabilities. Our defense mechanism tolerates arbitrary reads as we do not rely on secret information

Checker	stack	global	new/new[]	malloc family
CaVer	NO	PARTIAL	✓	NO
TYPESAN	✓	✓	✓	✓

Table 6.4: Allocation types tracked by checkers, see Section 6.5.1 for a detailed discussion.

that is hidden from the attacker.

6.4 Overview

TYPESAN is an extension to the Clang/LLVM compiler [81] that detects invalid `static_cast`s (i.e., all instances in the program where an object is cast into a different type without using an explicit run-time check through `dynamic_cast` or an explicit override through `reinterpret_cast`) in legacy C++ programs. Upon detection of an invalid cast, the program is terminated, optionally reporting the cause of the bad typecast. TYPESAN is a compiler-based solution and any C/C++ source code can be hardened, without modification, by recompiling it with our modified `clang++` compiler with the `-fsanitize=type` option and linking against the tcmalloc memory allocator [49] using the `-ltcmalloc` linker flag. As we show in Section 6.9, TYPESAN has reasonable performance for usage in production software.

Figure 6.1 presents an overview of TYPESAN. The *instrumentation layer* consists of hooks inserted by the compiler to monitor object allocations and potentially unsafe casts, as well as a static library containing the implementations of those hooks. To perform its task, this layer makes use of two services. The *type management service* encodes type information and performs the actual typecast verification. It includes type information for all the types that may be used at run time. The instrumentation layer uses it to look up type information for new allocations and informs it whenever a cast needs to be checked. Finally, the *metadata storage service* stores a pointer-to-type mapping and can look up the type information of an object about to be typecast. This service allocates a memory area to store the mapping at run time. It provides an operation to bind type information to a pointer and to lookup a previous binding for a pointer.

All mechanisms that explicitly protect from type confusion will incur two forms of run-time overhead: overhead for maintaining metadata (allocating and deallocating objects) and overhead for explicit type checks at cast locations. We designed TYPESAN to minimize the allocation/deallocation time overhead. C++ programs are heavily affected by allocator performance, as implicitly shown by how large projects tend to replace the standard memory allocator with high-performance allocators (tcmalloc and other allocators in Chrome, jealloc in Firefox). Many of the objects being allocated may also never be subject to downcasts, making it even more important to minimize the impact of type tracking on such objects.

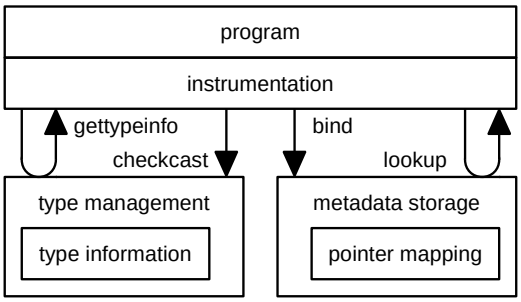


Figure 6.1: Overview of TYPESAN components.

In order to meet these requirements, TYPESAN relies on a clean, uniform front-end design that flags allocations and casts to the back-end system but introduces a completely different mechanism to track type information, called the metadata storage service in our design. Our metadata storage service builds on the memory layout and structure inherently provided by the allocator and uses this structure to reduce the access overhead ("aligning" objects with their metadata). Compared to existing disjoint metadata storage layers that use different forms of lookup functions from red-black trees to hashing for pointer range queries, our approach offers fast constant-time updates and lookups.

For the type checking instrumentation and related data structures, we use a design focused on simplicity and cache-efficient traversal. This design is effective even for workloads with an extremely high number of casting operations. Furthermore, for performance-sensitive applications, safe operations can be blacklisted or an approach like ASAP [131] can trade off security against acceptable overhead. This is another key motivation to minimize allocation-time overhead, as it does not scale with the number of instrumented casts in a program, acting as residual overhead instead.

Lastly, the instrumentation layer and the metadata storage service are connected by the instrumentation layer, which uses the Clang/LLVM compiler framework [81] to track allocations, instrument cast operations and extract type information. The instrumentation was designed with completeness in mind, following code patterns discovered in real-world programs as well as basic C/C++ constructs expected to be supported. Our instrumentation also allows full C-C++ inter-operability, a novelty compared to state-of-the-art solutions. This is important as some SPEC programs mix C-style allocation with C++ classes and browsers also use a mixture of C and C++ code.

6.5 Instrumentation layer

In this section we discuss the design of TYPESAN's instrumentation layer. The instrumentation layer interacts with the TYPESAN-hardened program by inserting hooks at relevant locations. We first consider the instrumentation of allocations, including the types of allocations we support to be able to track run-time object types with high coverage. Then, we discuss the instrumentation of typecasts to be able to introduce type checks.

6.5.1 Instrumenting allocations

TYPESAN adds a compiler pass to LLVM [81] to detect object allocations and insert instrumentation to store the corresponding pointer-to-type mapping. For each allocated object we store a pointer corresponding to the type layout of the object as a whole. However, keep in mind that downcasts in C++ might be applied not just to the allocated object pointer, but also to pointers internal to a given allocation range, specifically in the case of composite types (arrays, multiple inheritance, nested structs/classes). For example, an object of class `A` containing objects of classes `B` and `C` can be cast to `B*` using the original pointer while a cast to `C*` requires an offset to reference the correct member of class `A`. In this scenario, the type of the internal pointer differs from the type associated at the allocation time, which we need to account for in the design. For performance reasons, we chose to keep the instrumentation of the allocations as simple as possible and to defer handling composite types to the check operation itself (discussed in Section 6.6). This approach still introduces two additional requirements to our design. First, the metadata storage service must be able to retrieve the mapping for internal pointers (discussed in Section 6.7). Second, the checker needs access to the offset within the class definition corresponding to the internal pointer. To support the latter, we also track the base pointer of the allocation besides the type mapping. This design results in simple and low-impact instrumentation, using the metadata storage service to only store two pointers for each allocation in the program. In the following, we describe the allocation types we support as well as their motivation.

In C++, objects may reside in global memory, stack memory, or on the heap. These three kinds of objects have different lifetimes and we therefore instrument them differently. However, other than the location where we insert the hooks to instrument the allocation, the single uniform metadata storage service allows us to to treat objects in different memory areas equally. This simplifies our solution and improves performance because TYPESAN must not determine where a pointer lives before looking it up.

The initialization and reuse of the mappings in object metadata depend on the object's memory area. For instance, we can initialize the mappings for *global objects* once using global constructors. However, for stack and heap objects, we need a dynamic approach.

In the case of *stack objects*, we need to notify the metadata storage service to take control over the object. These objects are not put on the regular stack but are instead moved to a specific memory area where metadata tracking is enabled (see Section 6.7 for details on this operation). With this change, we can use the metadata storage service to create a mapping from the new object pointer to its metadata at allocation time. This design decision of moving the objects of interest to a separate location brings additional benefits from a tracking perspective, since the memory location occupied by a previously tracked stack object will only be reused for another tracked object. During allocation, the new object will overwrite the old metadata, removing it from the system permanently. This allows us to persist the metadata mapping after the lifetime of a stack object, removing the need for explicit cleanup.

A special class of stack objects (often ignored in existing solutions) arises when the program passes classes or structs by value as function arguments. To address this special case, TYPESAN uses the same approach applied by the current SafeStack implementation in LLVM, moving such objects to a separate stack[1].

Not all stack objects need tracking and we optimize our solution by omitting allocation instrumentation wherever we can prove that the program will never cast the allocated stack objects. To be conservative, we verify whether the function itself or the functions it may call perform any relevant casts. We assume that any indirect (including virtual) or cross-module function calls may perform downcasts, because for these cases we cannot statically determine that they do not. Using this approach, we reduce overhead without missing any checks. It is worth noting that our approach is more conservative than CaVer's, which optimistically considers that such callees never attempt casts within their respective call-graphs[2].

For *heap objects*, we add instrumentation after calls to the `new` and `new[]` operators as well as the traditional `malloc` family of allocation functions. Although C++ structs and classes are expected to be allocated using `new` (to ensure calls to the appropriate constructors), we observed that one of the four SPEC benchmarks with downcasts, uses `malloc/realloc` for allocating its C++ objects. Specifically, the soplex benchmark uses `malloc/realloc` to handle allocations within its core `DataSet` class, which acts like an object pool. Other classes, such as `SVSet`, maintain pointers to objects managed by a particular `DataSet`. As these pointers are also subject to downcasting, it is critical to track `malloc/realloc` in order to have type information available for checking. Tracking heap deallocation is not necessary as we built the metadata storage service to be robust and to clean stale metadata. This ensures that such metadata cannot influence type checks in case of an accidentally missed deallocation. More details can be found in Section 6.7.

While inferring the allocation type and array size is trivial for `new` and `new[]` (as it is part of the syntax), this is more complicated for the `malloc` family of functions. We traverse the intermediate representation code to look for cast operations applied to the resulting pointer to find the allocation type. This method might fail for type-

[1] `http://reviews.llvm.org/D14972`
[2] We reported this issue to the authors of CaVer.

agnostic allocation wrappers, but such wrappers can easily be added to the set of allocation functions which we track. Array allocations can be tricky when trying to infer the element count from a `malloc`-like call-site, but our tracking scheme was designed to be agnostic to array sizes, thus mitigating potential issues. In practice we found no coverage issues when evaluating TYPESAN against SPEC, showcasing our ability to track relevant heap objects with our solution.

An interesting point in heap allocations is support for allocations within the standard template library (STL), which countless applications use for their core data structures. Luckily, STL's template-based design means that all the code related to data structures is located within headers included into every source file. This includes all their allocation wrappers, which are also templated and instantiated on a per-type basis. We confirmed that our instrumentation correctly picks up the allocations within the STL data structures and we successfully check the downcasting operations applied to the individual elements.

6.5.2 Instrumenting typecasts

Whenever TYPESAN encounters a downcast operation (from a base class to a derived class), it inserts a call to our typecasting verification function. Such a cast is present in the code either when performing a `static_cast` operation between appropriate type or when using an equivalent construct such as static C-style casts. In practice, downcasting can exhibit two types of behavior and we optimize our checker to support each one specifically. In the general case, the result of the static cast is the source pointer itself (with no added offset), but with a new source-level type. This happens when the source base type is the primary base class of the derived type, which is always the case when casting without multiple inheritance. In this case, the TYPESAN instrumentation calls the checker with the value of the pointer and an identifier corresponding to the destination type. If classes use multiple inheritance it can happen that a cast operation occurs from a secondary base class to a derived type. In this scenario, a negative offset is added to the source pointer as transformation from an internal pointer (to the secondary base) to the base pointer of the object. The checker needs information about the resulting pointer to infer the appropriate offset within the structure layout, but in case of type confusion the negative offsets might make a valid pointer go out of bounds, making it impossible to infer the appropriate type information for the object pointed to. For this reason, TYPESAN calls a second version of the checker in this instance, which takes both the source and destination pointers as well as the type identifier for the resulting type.

6.6 Type management service

TYPESAN manages metadata on a per-allocation basis. Every block allocated by a single `malloc` call, `new` operator, global variable, or stack variable is associated

with at most a single pointer to a type information data structure. This data structure therefore encodes all permissible casts using pointers pointing into the allocated block. Any object can be cast back to itself using the original pointer, but composition and inheritance create additional opportunities for casts. For example, a pointer to an object can be cast to any of its base classes and a pointer to a field of an object can be cast to the type of the field (and transitively to any type on the inheritance chain). In this section, we discuss the data structures used to encode this information.

The type management service is responsible for associating type layouts with allocation sites and using these layouts to validate downcast operations. Type checking at its core can be divided into two steps. The first one is the ability to infer the allocation-time type associated with the pointer resulting from the typecast. This is the most derived type associated with the particular offset (from the pointer to the allocation base) within the original allocation. Once this information is known, the second part of the type check involves the comparison of the allocation-time type with the type specified in the cast. The layout of the latter must be compatible with the former for the cast to be valid. The data structures employed by this service share the same purpose as the `THTable` structure in CaVer, but we further optimize the type checks by dividing it in in two phases.

In the following sections, we describe the data structures TYPESAN uses to perform these operations.

6.6.1 Type layout tables

Type layout tables describe each relevant offset within an allocation to enable fast lookups of the offset-specific type information during a check. Specifically, a type layout table is a list of mappings of unique offsets to data fields corresponding to nested types. The list (array) starts with an entry for offset 0 containing the unique hash corresponding to the type as a data field. The layout tables incorporate nested types in one of two ways. As a first option, the nested types can be flattened into a type layout for good cache efficiency during traversal. Alternatively, they can be separated via an indirection for better space efficiency. Flattening a nested type involves injecting an offset-adjusted copy of its type layout into the containing type, where its type layout is copied and adjusted into the layout of the containing type.

An avid reader may notice that flattening can invalidate the property mentioned earlier that offsets in the type layout table are unique. After all, a nested class might occur at offset 0 (for example with primary base classes). This is where the second part of our type check, the layout compatibility check, comes into play. A class which includes a nested class at offset 0 is practically layout compatible with the latter. Intuitively, if the next type class has another class at offset 0 of object, then we can always use the object as representative for both types. TYPESAN tracks this relationship in *type relationship tables* (see Section 6.6.2). Thus, the type layout table only needs to track the (unique) "derived-most" type for matching offsets.

As mentioned earlier, flattening is not compulsory, since the type layout table also supports indirection. In this mode, the data element of a particular entry includes a pointer to the type layout table corresponding to the nested type. In addition, TYPESAN adds a sentinel element to the type table to mark the end of the nested type. This allows the traversal code to infer quickly whether or not it should follow the indirection. It skips the sentinel element if its offset overlaps with an existing entry in the table to maintain uniqueness.

Flattening generally improves performance at the cost of space. While TYPESAN mostly uses the flattened mode, it uses the non-flattened mode to generate an efficient array abstraction as the array length may be dynamic (in the lack of a way to optimize for performance we optimize for space). In particular, it replaces each nested array with a single indirect entry to the type layout table of the array element type, allowing TYPESAN to support nested arrays of any size, without degrading checking speed.

The property of enforcing unique offsets in the type layout table allows us to implement efficient traversal by ordering the entries by offset. During indirection, the type management service updates the offset that is being searched to match the follow-up type layout table (which is just a subset of the original type). In the case of a nested array, it updates the offset to represent the offset into the corresponding array element, instead of the array itself—using the array-stride as input. This is supported, by including the overall size of the type (including requested alignment) as part of the type layout table.

When the instrumentation adds metadata at an allocation site, it simply requests a type layout table that corresponds to the type that is allocated. Type layout tables are generated once for each type, by recursively traversing the type information in LLVM to find all nested elements. For potential array allocations, it marks the pointer to the type layout table to signal additional processing of the offset. This processing is identical to how we deal with nested arrays. Accidentally marking an allocation as being of type array does not affect correctness, it just involves a couple of extra instructions being executed during type lookup. As such our static analysis does not have to be complete as long as it is conservative in identifying allocations of single elements.

6.6.2 Type relationship tables

In the second stage of the type check we check for raw pointer compatibility between the type identified for the pointer and the type defined in the cast. Such compatibility typically happens between derived classes and their primary base class. As mentioned earlier, another case of compatibility happens between unrelated types if one of the types is nested in the other at offset 0.

Furthermore, CaVer defined the concept of phantom classes: derived classes with no additional members compared to their base class. Sometimes the program downcasts base classes to such phantom classes, resulting in benign bad casts. Thus we

also include phantom classes in our compatibility model. Using the compatibility model, we generate a set of type hashes corresponding to the compatible types for each class in the program and refer to it as a *type relationship table*. Once TYPESAN has extracted the type of a pointer from the type layout table, it checks the corresponding type relationship table for the membership of the type defined in the cast. The operation needs to find an exact match to verify a cast. If it finds no match, it reports an error of a mismatched type.

Currently we implement sets as simple vectors, with hashes ordered according to the type hierarchy. We found this solution to be adequate in terms of speed, but we can easily replace it with alternative set implementations, such as the fast LLVM bitset variant. Doing so is easy as the type relationships table is conceptually nothing more than a set as a result of the split of the type information into separate layout and relationship tables.

By having the phantom classes be first-class members of the type relationship tables, we ensure uniform support for them without performance degradation. In contrast, the publicly released CaVer code requires a type check restart for every phantom class if normal inheritance fails to find a match.

6.6.3 Merging type information across source files

Generating large amounts of type information may necessitate merging across different source files—an expensive and potentially difficult operation. We rely on the One Definition Rule (ODR) in C++ to minimize the need to merge information across the project. ODR states that every class definition across all source files linked together needs to be identical. C++ also requires nested and base types to be fully defined in the source files that use them. As a result, the type layout information for the same type within different source files is always identical. The same is true for the type relationship tables—except their phantom class entries. Since the phantom classes represent derived classes, the set of phantom classes can easily change from one source file to another, and merging may be necessary. TYPESAN uses a strategy where it only needs to merge these entries in the type relationship tables to minimizing the merging cost. Any program violating the ODR would trigger error reports in TYPESAN. This would be correct, since violating the ODR is type confusion.

6.7 Metadata storage service

In this section, we discuss the metadata storage service, which handles storage of metadata at run time. This service allows us to map from object base addresses to type layout tables at run time. Key requirements for our metadata storage service are (i) fast update operations and (ii) range-based lookups (to support interior pointers due to composition). Related work [84; 7] has used alignment-based direct mapping schemes to track complex metadata, relying on dedicated custom memory

allocators. Such systems often run into problems for stack-based memory allocations [84] where the allocator has no detailed knowledge of allocations. As a result we designed METAlloc [66], a novel object tracking scheme, based on variable compression ratio memory shadowing, which solves these issues and allows us to have an efficient and uniform metadata storage service for all allocation types.

Variable compression ratio memory shadowing relies on the assumption that all objects on a certain memory page share the same alignment. A uniform alignment guarantees that every alignment-sized slot within the page corresponds to a single object, enabling the tracking of metadata at the level of slots instead of objects, while preserving correctness. Each page can thus be associated with an alignment and a metadata range, including as many entries as the number of alignment-sized slots in the page. Such a mapping simplifies storage of metadata and allows us to assume a certain layout for objects on a per-page basis. Given a page table-like infrastructure, the mapping allows finding metadata corresponding to any particular pointer in constant time, by using the alignment to look up the slot index and to retrieve the metadata stored for the appropriate slot. This mapping also mirrors traditional page tables as the alignment and the base address of the metadata range can be compressed into a single 64-bit value (since pointers only require 48 bits). We call this data structure the *metadata directory*. Figure 6.2 shows the mapping operation from any pointer to the corresponding object metadata.

An update operation with this metadata results in finding the metadata for the base address of the object and then updating all entries which correspond to the object range. The number of entries which need to be updated is the number of alignment-size slots that the object spans across, making it critical to select the largest possible alignment to improve update performance. This is where the variable compression ratio comes into play, with large objects having larger alignments, thus their metadata is compressed relative to their size. The system also works with the default 8-byte alignment of objects in existing systems, but the update operation would end up too costly for stack and heap objects. Using alignments which are too large can also generate increased memory fragmentation resulting in unnecessary performance overhead, making it critical to select the most appropriate alignments.

In the case of global memory, the overhead introduced by the update operations rarely affects the performance of a running program, thus we decided to leverage the existing 8-byte alignment applicable for global objects. We update the metadata directory entries to track all loaded sections whenever we detect that a new shared object has been loaded into the address space of the program.

For heap allocations, tcmalloc [49] (the allocator used by the Chrome browser and other Google products) already ensures that every memory page under its control contains only objects of the same size-class and alignment. It enforces this property to efficiently generate free lists, thus ensuring our assumptions for free, without needing to perform any changes to the allocation logic. We only extended tcmalloc to track the metadata directory entries whenever a memory page is associated with a certain size-class, which happens rarely in practice.

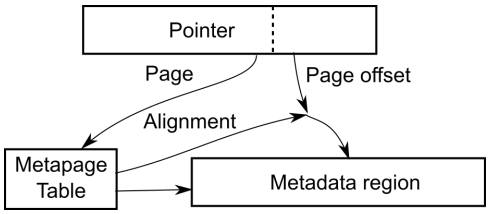


Figure 6.2: Mapping from a pointer to a metadata entry. The page component is used to look up the start address of the metadata region and the alignment. While the latter together with the page offset is used to compute the offset within the metadata region.

Stack allocations are challenging, as they can be subject to ABI restrictions. We mitigate this limitation by moving relevant objects to a secondary stack similar to the operating principles of SafeStack [80]. SafeStack is effective at moving dangerous stack allocations to a secondary stack with practically no overhead and minimal impact on application compatibility. We use the instrumentation layer, as mentioned earlier, to tell the metadata storage service about stack objects, which are then moved to a secondary stack tracked by the metadata directory, where we enforce a 64-byte alignment for each object. ABI restrictions are not applicable, since all tracked stack objects are local variables, whose location can be freely chosen by the compiler. The 64-byte alignment is reasonable as we only move a small subset of stack objects, thus overall memory fragmentation of the program is limited.

As mentioned earlier in Section 6.5, deallocation of heap objects is handed internally by the metadata storage service and no extra instrumentation is needed. While the stack contains only tracked objects, thus metadata will always be up to date with allocations, `tcmalloc` does manage all heap objects, including untracked ones. As such, we extended `tcmalloc` to conservatively assume that every new allocation is untracked and to clear stale metadata associated with any new allocation if it detects such. This approach minimizes the overhead when metadata is used sparingly, while ensuring that stale metadata can never affect untracked allocations.

Moreover, our solution is not affected by thread concurrency. The metadata directory is only updated when memory is mapped in from the system. At this point all the entries read/written during this operation are the ones corresponding to the allocation range, which is still in sole control of the running thread. The metadata entries are also updated only during object allocation and the entries written are unique to the allocation range, thus they do not interfere with concurrent lookups. The update operation depends only on the metadata directory entry corresponding to the pointer itself, which cannot be subject to a concurrent write.

6.8 Limitations

Our approach is based on an LLVM-instrumentation pass that reasons on the clang and LLVM IR level, therefore source code in either C or C++ is required. Any allocations or casts in assembly are not supported.

As stated in our treat model (and similar to related work), we assume that the attacker has no unrestricted write primitive at their disposal. We therefore do not protect our metadata against malicious writes. Any metadata protection mechanism is orthogonal to this work and equally applies to other protection systems which complement TYPESAN.

To support combined protections, TypeSan deliberately imposes as few restrictions and changes in the memory layout as possible. It already integrates with SafeStack, a fast stack protection solution offered by compilers today. Similarly, while TypeSan makes a design assumption about the heap allocator, it is compatible with arguably the two most commonly used custom memory allocators: tcmalloc (Chromium) and jemalloc (Firefox and Facebook). When combined with other memory safety solutions, TypeSan can preserve metadata integrity by construction. When deployed standalone, our design is amenable to existing low-overhead metadata protection techniques, such as APM [56] or write-side SFI (e.g., pointer masking). The overhead of such techniques is amortized across all defense solutions. For example, if we employ SafeStack and we expect SafeStack (already in clang) to be adequately protected moving forward due to recent attacks [56; 100], TypeSan can benefit from the same metadata protection guarantees at no additional cost. Note that TYPESAN tolerates arbitrary reads as we do not rely on secret information hidden from the attacker.

Custom memory allocators (CMAs) can prohibit TYPESAN from appropriately tracking heap allocations. Unfortunately, this is a fundamental limitation for instrumentation systems which rely on object information tracking. TYPESAN uses `tcmalloc` as the back-end allocator. This is a suitable replacement for other general purpose allocators, but objects allocated within CMAs (such as pool or SLAB allocators) will not be tracked by our system.

6.9 Evaluation

To show that TYPESAN achieves higher coverage and lower overhead than previous solutions, we evaluated our prototype using a number of demanding CPU-intensive (and cast-intensive) workloads. We test Firefox because browsers are a common target for attackers of type confusion vulnerabilities, given the fact that they are usually written in C++ for performance reasons and have a large attack surface because of the fact that they provide a substantial API to foreign Javascript code. We benchmarked Firefox using the Octane [57], SunSpider [58], and Dromaeo [46] benchmarks. Octane and SunSpider focus on Javascript performance, while Dromaeo has

subtests for both Javascript and the DOM. Moreover, we implemented our own microbenchmarks to isolate the overhead and report worst-case figures for TYPESAN and existing solutions. In addition, we run the SPEC CPU2006 C++ benchmarks, which all heavily stress our allocator instrumentation and some (e.g., dealII and omnetpp) our typecast instrumentation.

In our evaluation, we consider a number of different system configurations. Our baseline configuration compiles with Clang 3.9 [81] at the default optimization levels. The baseline is not instrumented but does use the tcmalloc [49] allocator to report unbiased results, given that tcmalloc generally introduces a speedup that should not be attributed to TYPESAN. In addition to the baseline, we have the TYPESAN and TYPESAN-res configurations. The former instruments every possible cast while the latter does not instrument any casts. The TYPESAN-res configuration shows to what extent a system like ASAP [131] can reduce the performance impact of our instrumentation (trading off security) when only a small performance budget is available. We ran our benchmarks on an Intel Core i7-4790 CPU with 16 GB of RAM, using the Ubuntu 15.10 Linux distribution.

6.9.1 Performance

Microbenchmarks

To verify that TYPESAN provides low allocation-time and typecast-time overhead, we created microbenchmarks that measure how long these operations take, both on the stack and on the heap. To compare our results against state-of-the-art solutions, we compiled CaVer [84] from source [82] and configured in the same way as TYPESAN—except that we do not use tcmalloc since CaVer ships with its own custom allocator. To prevent the target operations from being removed by optimizations, we switched to the `-O1` optimization level for this experiment. To isolate the overhead, we measured the impact of (i) the number of allocated stack objects and (ii) the object size. The former is important since CaVer tracks stack objects with red-black trees, whose performance degrades with the number of objects. The latter is important since TYPESAN needs to initialize multiple metadata entries for large objects, incurring more overhead.

Figure 6.3 depicts the impact of the object size on allocation performance when no other stack objects are present. Allocating an object on the stack is almost instantaneous for the baseline and takes a fixed but long time for CaVer. For TYPESAN, allocation time on the stack is proportional to the object size as multiple metadata entries need to be initialized for large objects. However, even for objects as large as 8KB, TYPESAN is still faster than CaVer. Small objects up to 128 bytes take only 0.5ns extra to allocate with TYPESAN, while CaVer adds at least 48.8ns even for these small (and common) allocations. On the heap, allocation time grows linearly with the allocation size in all cases. Overall, TYPESAN is close in performance to the baseline while CaVer adds considerable overhead. For heap allocations up to 128

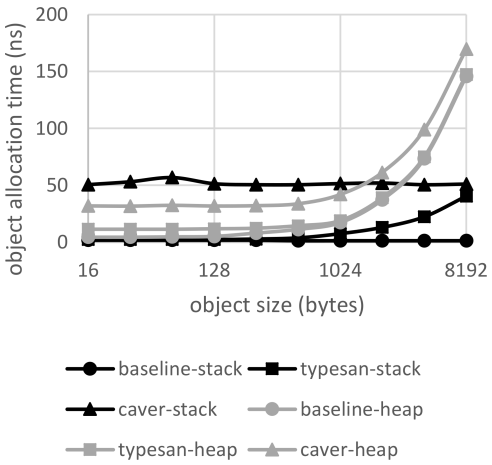


Figure 6.3: Allocation performance as a function of allocated object size.

bytes, TYPESAN adds at most 7.0ns overhead while CaVer adds at least 26.7ns.

We believe that it is unlikely for programs to frequently allocate large, bloated classes without using them and thus hiding the allocation overhead. As further mitigation to the scaling based on the stack object size, it is possible to extend TYPESAN to use additional secondary stacks with increased alignments for such large classes. These results support our claims that TYPESAN is particularly suitable for applications that allocate many objects, especially on the stack.

Figure 6.4 and Figure 6.5 show the impact of the number of allocated stack objects on allocation and typecast performance (respectively), using an object size of 16 bytes. The overhead patterns are in the same region for both scenarios. CaVer's overhead on the stack increases with the logarithm of the number of objects (due to the use of red-black trees) while TYPESAN's does not depend on the number of allocated objects. Even at relatively low allocation counts, CaVer's stack allocations are much more expensive than TYPESAN's.

For every typecast, in turn, TYPESAN adds an overhead of only 3.8ns, regardless of whether the object is allocated on the stack or on the heap and regardless of the number of allocated objects. CaVer's typecast overhead is higher in all cases. In particular, the heap overhead is a constant 13.6ns, while the stack overhead starts at 11.0ns and increases with the number of allocated objects.

Performance overhead

Table 6.5 reports our performance on the SPEC CPU2006 C++ benchmarks [68] and Firefox. The first four SPEC benchmarks perform static typecasts while the others do not. In the latter case, the overhead stems from TYPESAN having to still track objects that cannot be statically and conservatively proven not be typecast during the execution. In the default configuration, overheads range from negligible to moder-

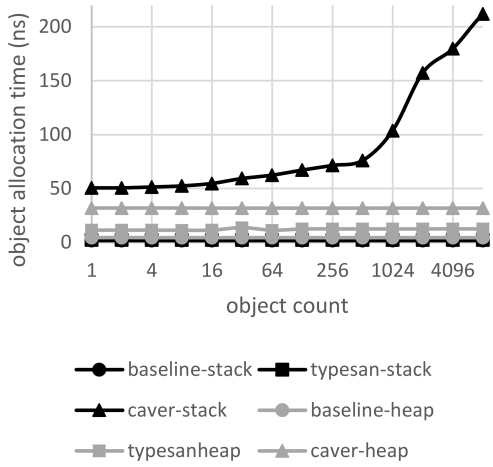


Figure 6.4: Allocation performance as a function of allocated object count.

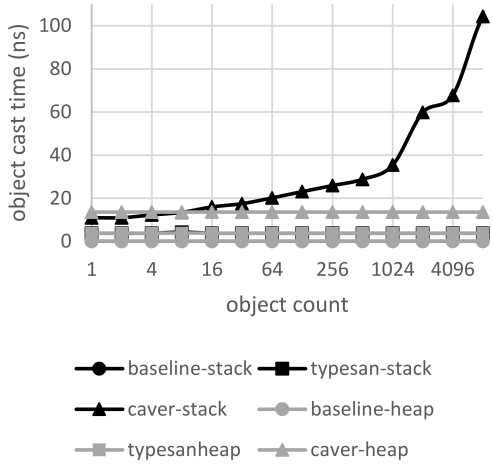


Figure 6.5: Typecast performance as a function of allocated object count.

ate and the overheads on the benchmarks reported by CaVer [84] are much lower. In particular, CaVer reported four times our overhead (29.6%) on xalancbmk and 20.0% on soplex while ours is negligible. Povray stands out for having high overhead despite its lack of casts. This is mostly due to the many stack objects allocated in a recursion between the functions `Ray_In_Bounds` and `Intersection`. Other than this special case, overheads are lowered by reducing the number of checks (a la ASAP [131]). The negative overhead for namd may be explained by variations in memory layout [94]. For example, in all but one case, our TYPESAN-res configuration can greatly bring down the overhead. The overhead for Firefox is unfortunately somewhat higher, especially for the Dromaeo DOM workload. The high overload is

	casts	TypeSan		TypeSan-res	
dealII	yes	30.8	(0.2)	3.6	(0.1)
soplex	yes	1.8	(0.7)	1.5	(0.8)
omnetpp	yes	27.2	(1.7)	2.4	(3.0)
xalancbmk	yes	7.1	(0.4)	4.4	(0.3)
namd	no	-0.6	(0.1)	-0.5	(0.1)
povray	no	23.9	(0.3)	22.6	(0.2)
astar	no	-0.2	(0.5)	0.1	(0.5)
geomean	yes	13.2		6.4	
geomean	both	12.1		4.6	
ff-sunspider		40.6	(1.4)	11.4	(1.1)
ff-octane		18.6	(5.1)	2.8	(0.9)
ff-drom-js		12.4	(1.5)	4.0	(1.7)
ff-drom-dom		71.2	(1.5)	43.5	(0.6)
geomean		33.9		14.3	

Table 6.5: Performance overhead on the SPEC CPU2006 C++ benchmarks and Firefox (%), stdev in parentheses.

most likely due to the fact that Firefox performs many object allocations, especially on the stack. On average, however, our overhead is close to half of the overhead reported by CaVer (64.6%). The results for the TYPESAN-res configuration show that this overhead can be reduced even further by selectively instrumenting casts. Note that our coverage on Firefox is considerably lower than on the other benchmarks mostly due to the use of CMAs (though still much higher than the competition, see Table 6.8), so extending our solution to cover the remaining casts could increase the overheads reported here.

Memory overhead

Table 6.6 reports our memory usage on the SPEC CPU2006 C++ benchmarks and Firefox, measured in terms of binary size (static) and the maximum resident set size (dynamic). While TYPESAN generally introduces nontrivial memory overhead, we believe this is worthwhile (and acceptable in practice), given the security it offers with negligible performance overhead. Moreover, compared to CaVer [84], our relative run-time memory overhead is much lower for the two SPEC benchmarks they considered (9% vs. 56% on xalancbmk and 1% vs. 150% on soplex). It is unfortunately impossible to compare our memory overhead on Firefox as we measured a different version than the one reported by CaVer. The negative memory overhead for Dromaeo-DOM may be explained by the fact that this benchmark runs for a fixed amount of time, completing fewer runs when the browser is slowed down through our instrumentation. Our memory overhead is mostly due to the metadata storage service itself. As future work, it is possible to share this infrastructure and its memory overhead with other defenses that maintain per-object metadata (e.g., bounds

	binary size			resident set		
	base	ts	inc%	base	ts	inc%
namd	0.3	0.6	81.6	50.9	56.9	11.7
dealII	3.1	5.2	69.3	818.6	1453.1	77.5
soplex	0.4	0.8	112.7	560.9	568.4	1.3
povray	1.0	1.4	45.2	8.7	18.8	117.4
omnetpp	0.6	1.2	90.4	157.8	224.5	42.3
astar	0.0	0.3	597.1	310.4	314.4	1.3
xalancbmk	4.1	7.6	85.8	451.3	492.7	9.2
geomean			118.0			31.7
ff-sunspider	159.1	318.6	100.2	491.1	928.2	89.0
ff-octane	159.1	318.6	100.2	844.4	1534.4	81.7
ff-drom-js	159.1	318.6	100.2	572.2	1005.8	75.8
ff-drom-dom	159.1	318.6	100.2	4232.0	4015.7	-5.1
geomean			100.2			19.9

Table 6.6: Memory usage for the SPEC CPU2006 C++ benchmarks and Firefox (MB), ts=TYPESAN.

checking), reducing the memory usage of the combined system.

6.9.2 Coverage

Typecast coverage test suite

To test the correctness of TYPESAN and future type-confusion detection systems, we developed a test suite for a wide range of different code constructs that might affect typecast sanitization. The test cases covered by the test suite are inspired by our extensive experience with real-world C++ programs. Our test suite is available as part of the open source repository of TYPESAN to help future researchers in testing their systems. The test suite verifies correctness using three different dimensions: *allocation type*, *composition type*, and *cast type*. The test suite allows different configurations from each dimension to be combined and tested simultaneously.

Every test allocates an object of type `AllocationType` using the desired configuration. It then sends a pointer to the allocated object to a function, which derives a pointer to a member of type `BaseType` nested into `AllocationType` with the desired composition type configuration. Finally, we downcast the pointer to a type derived from `BaseType`, called `DerivedType`. We implement the functions in different source files to uncover any potential bugs in interprocedural cross-module static analysis in the checker. Other than false negatives, the test suite also checks for false positives, by replacing the `BaseType` with a derived type of `DerivedType` in `AllocationType`.

Our test suite considers the following allocation types: stack object, stack array, struct argument passed by value, global object, global array, new / new[] (including

	allocations			casts	types
	heap	stack	global		
dealII	322	5,231	1,125	716	1,238
soplex	40	447	161	2	311
omnetpp	441	85	623	449	568
xalancbmk	414	3,216	2,263	2,688	1,768
namd	10	18	4	0	16
povray	36	255	200	0	257
astar	13	11	7	0	18
firefox	2,458	185,908	42,710	68,369	38,764

Table 6.7: Instrumentations.

overloaded operator), and malloc/calloc/realloc (including arrays). For array types, we also consider multidimensional arrays where possible. We consider the following composition types (the relationship of `BaseType` with respect to `AllocationType`): equivalence, nested, nested array element, and inheritance (primary, secondary, and virtual). Finally, we support the following cast types: from primary base to derived type, from secondary base to derived type, and from primary base to a phantom class of the real type.

TYPESAN successfully passes all combinations of these configurations, showcasing our coverage on a wide range of code constructs. For comparison, we also tested CaVer [82] and found that it only passes the following allocation types: global object and new / new[] (including overloaded operator). CaVer also reported false positives when nested arrays were used for the composition. We contacted the authors about all the tests that failed, but we have not received an explanation or a fix for these issues.

Coverage on benchmarks

Table 6.7 shows the number of code locations where we insert instrumentation. Note that the information about allocations is based on the source code because it cannot be easily recognized in the binary due to inlining. The number of cast checks and types are based on the final binary to make them comparable to CaVer [84]. The number of checks in the source code is considerably lower (for example 19,578 for Firefox), presumably due to optimizations that cause code to be duplicated. Compared to CaVer, we insert slightly fewer checks in Firefox, more in Xalanc and the same number in Soplex. This may be due to differences in versions or compiler (settings). We generate more type information than CaVer on Firefox and Soplex, but less on Xalanc. This may be due to differences in representation of type information. In all cases, we insert more instrumentation than UBSan does [84].

Table 6.8 shows the typecast coverage of TYPESAN compared to state-of-the-art solutions, CaVer [84] in particular. Coverage percentages are computed as the fraction of non-null casts correctly checked by the system—missing checks are due

	allocations		casts				
	TypeSan	CaVer	non-null	TypeSan	%	CaVer	%
dealII	597m		3,596m	3,596m	100.00	0	0.00
soplex	21m	1,058	209k	209k	100.00	0	0.00
omnetpp	264m		2,014m	2,014m	100.00	0	0.00
xalancbmk	4,538m	278k	284m	254m	89.52	24k	0.01
ff-sunspider	463m		293m	92m	31.43		
ff-octane	967m		991m	122m	12.35		
ff-drom-js	15,824m		12,976m	3,032m	23.37		
ff-drom-dom	301,540m		46,961m	21,253m	45.26		
ff-total	318,793m	15,530k	61,222m	24,500m	40.02	1,077k	0.00

Table 6.8: Typecast coverage.

to inability to correctly track the corresponding object. As such, it is an indication of the security provided by the system. TYPESAN reports over 89% coverage on each of the relevant SPEC benchmarks, and 100.0% on all but one. Unfortunately, coverage is not as high on Firefox. We have found that this is due to the widespread use of pool allocators, which violate the assumption made by our system (as well as other object tracking systems) that objects are allocated individually. This issue could be solved by modifying Firefox to allocate objects directly. This may be viable performance-wise due to the allocation performance offered by tcmalloc. While CaVer does not report per-benchmark results, they report a total of 1,077k verified casts. As such, our coverage is approximately more than 300,000 times as high. For both SPEC and Firefox, Table 6.8 shows that we track many more allocated objects than CaVer. This explains that we are able to check more casts despite a similar number of instrumentation sites—casts can only be checked if type information metadata was stored at allocation time. As shown in Table 6.8, we provide security far superior to the current state of the art while improving performance at the same time.

6.10 Related work

To the best of our knowledge, UBSan [105] and CaVer [84] are the only other systems that perform verification at cast time like TYPESAN. Our system is inspired by CaVer and has considerable similarities with it. In particular, it shares the same benefits with regard to UBSan: we do not rely on run-time type information (RTTI) and therefore we can handle non-polymorphic classes and protect binaries without the need for manually maintained blacklists. Moreover, as we have shown in our evaluation, we introduce less overhead than CaVer, which in turn has shown by its authors to be more efficient than UBSan.

Compared to CaVer, we have a similar instrumentation layer based on an LLVM instrumentation pass, but we have completely redesigned the metadata storage mechanism. In particular, we use a *uniform* variable compression ratio memory shading

scheme with off-the-shelf allocation strategies, rather than a purpose built custom memory allocator of the heap and the red-black trees used for the stack in CaVer. Our approach is more efficient for both insertions (object allocations) and lookups (typecast checks) because it does not require identifying the type of the allocation (due to its uniform nature) and it does not incur the significant and non-linear overhead that red-black trees bring to trivial stack allocations. Moreover, our solution is not affected by thread concurrency. This is a major simplification compared to CaVer, which uses per-thread red-black trees.

Another solution that achieves similar goals as ours is preventing calls through incorrect virtual method tables (vtables). For example, Bounov et al. [19] present an approach that can efficiently verify for each virtual call that the vtable is valid for the static type through which the call is performed. This mitigates some type confusion vulnerabilities, but such solutions cannot protect non-polymorphic classes because they do not have vtables. Moreover, this solution only detects type confusion when the object is subjected to a virtual call, thus missing potential memory corruption from a mismatched layout in other parts of the code. Clang CFI [29] uses such a system to check cast operations involving polymorphic classes, but there is no publicly available evaluation of the system and it is still restricted to a subset of downcast operations.

On binaries without source code, Dewey and Giffin [40] show how data flow (reaching definition) analysis may help to determine bounds on vtables and detect type confusion statically by ensuring that a virtual function call does not stray beyond the bounds of the vtable. As noted by the authors, their analysis is prone to false positives and false negatives and therefore more suited to reducing the number of type confusion bugs prior to deployment.

Finally, CFI [3] and other advanced protection mechanisms for forward edges in C++ programs [127; 130; 129; 80] limit the wiggle room that attackers have to divert control via indirect control transfers. However, as type confusion is mostly a data problem, such solutions only address it partly. Similarly, VTable protection schemes [140; 63], may check the types of virtual calls or the sanity of vtable pointers, but do not prevent the misuse of type confusion in general.

6.11 Conclusion

Type confusion vulnerabilities play an important role in modern exploits as shown in recent attacks against Google Chrome or Mozilla Firefox. Existing solutions that detect type confusion exploits are (i) incomplete, missing a large number of typecasts and (ii) prohibitively slow, thereby hindering general adoption.

We presented TypeSan, an LLVM-based type-confusion detector that leverages an optimized allocator to store metadata in an efficient way to reduce the overhead for updating metadata. Building on several optimizations for both the underlying type checks and the metadata handling, we reduce the performance overhead by a

factor 3–6 compared to the state of the art. Our performance figures suggest TypeSan can be used as an always-on solution in practical settings. In addition, TypeSan is complete and no longer misses typecasts on either the stack or between C and C++ object interactions. As we show in the SPEC CPU2006 benchmarks, such interoperability issues between programming languages cause prior work to miss a large number of casts.

7 Conclusion

Despite the best efforts of researchers to eliminate or mitigate software vulnerabilities using a wide range of techniques, attackers still seem to possess the creativity and technical prowess to overcome said mitigations in practice. As such, researchers have to continuously innovate and adapt to keep one step ahead in this arms race. Vulnerabilities can be tackled at different stages of software evolution, but it is important to understand how these options are complementary to each other, instead of being exclusive. In practice the best defenses are the result of layering multiple mitigation techniques on top of each other, to minimize the chance of a successful exploit. This dissertation makes contributions to such an end-to-end defense scenario, investigating the effectiveness of existing techniques and proposing novel alternatives wherever major deficiencies are discovered. In the following, we summarize the key results of this dissertation.

1. ***Code artifact prioritization for security***. We presented the potential for *security prioritization*, an alternative idea to comprehensive testing, which aims to prioritize the testing effort around code artifacts with the highest chance to contain vulnerabilities. Typically such prioritization occurred as part of white-box testing, with manual input from developers, but we showcase the possibility to automate the process using source analysis. This idea lays the ground for effective modular testing, by automatically inferring the need to test certain individual code artifacts and the potential to avoid other artifacts.

2. ***Guided fuzzing***. We presented *guided fuzzing*, a new technique to restrict gray-box fuzzing to certain pre-selected code artifacts. This process improves the scalability of fuzzing significantly, by splitting the exponential state-space into manageable chunks related to the individual code artifacts. Since typical software carries a lot of dependencies between its different components, guidance has to occur on both the data and control planes to be effective. We implemented data-

flow tracking to learn about the data dependencies between the program inputs and the individual code artifacts. Guided fuzzing can leverage this dependency information to restrict the input space, which needs to be explored for this particular iteration. Furthermore we refined the guidance process, by implementing a learning mechanism to directly infer information about control dependencies. This allowed for further restrictions on the state-space via the control statements, bringing the desired scalability to guided fuzzing.

3. ***Pointer-structure reversing***. We presented a novel technique to detect and classify pointer-based data structures in binary programs without the use of debug or symbol information. These data structures include linked lists and a range of balanced/unbalanced trees for instance. The proposed technique offers a great deal of flexibility on two axis. First of all, it extracts course-grained type information based on usage information, making it independent of type reversing systems and their potential limitation. Furthermore, it uses a series of normalization stages to deal with variations in implementation details, while keeping the final classifier simple and easy to reason about.

4. ***Correctness evaluation for control-flow integrity***. We presented a novel white-box testing framework for control-flow integrity solutions targeting vtable-based call-sites within C++ code. The need for this framework arose from the prominence of this protection mechanism in production compilers these days (already part of GCC and in the works for Clang). The framework allows developers to check if their protection variants respect all the possible corner-cases within the C++ semantics and thus will work for any well-formed C++ program. It also checks if the protection restricts the control-flow options to only the edges required by the language semantics. Solutions passing the latter check are deemed to be optimal as further restrictions are not possible without the potential to break program execution. Solutions which fail this check can then be compared against the reference solution to evaluate their numeric effectiveness in practice.

5. ***Optimal control-flow integrity for C++***. We presented a novel design for control-flow integrity applied to vtable-based call-sites within C++ code. We used the mistakes learned from existing systems as discovered by the proposed testing framework to gain better understanding of the problem and to refine the original designs. The proposed alternative fixes all the limitations we discovered, offering optimal protection, while still ensuring full compatibility with the C++ semantics. The run-time overhead is also reduced as part of the proposed changes, a rare side-effect when increasing the strength of mitigation. These properties make the new design applicable to replace the existing code within the GCC compiler, beating out the existing code in all key characteristics.

6. ***Efficient metadata tracking***. We presented variable compression-rate metadata tracking a new technique to track object-level information during execution with

low memory and run-time overhead. It relies on recent developments in mainstream memory organization, to avoid having to include a specialized heap allocator or without the danger of break existing ABIs. The metadata tracking enables a wide-range of instrumentation techniques, including bounds checking, type checking and dangling pointer detection, which seem more and more necessary in an effort to counteract recent data-only attack. Our evaluations show that the run-time overhead scales well with increased metadata sizes, making the technique particularly effective in applications requiring complex metadata to function.

7. ***Verifying type safety for C++***. We presented a novel approach to validate potentially unsafe casting within C++ programs using the previously discussed metadata tracking as a backbone. The approach shows great potential with very low run-time overhead and no sacrifice to the desired coverage profile. The efficiency of the metadata tracking enables further reduction of the overhead for applications, where coverage can be sacrificed to reach a particular performance goal. The proposed design also represents a strong first step towards complete type safety for C++ programs, as all the building blocks are now in place to enforce it.

Future Directions

The problem of system security is never considered to be solved as malicious actors always find new and creative avenues for their attacks. The techniques presented in this dissertation are aimed to improve the state-of-the-art, but continued research effort is required to stay one step ahead of attackers. Hopefully one day all our core infrastructure is migrated to safe programming languages to at least avoid memory corruption attacks, but until that point we are required to invest into detection and mitigation systems as discussed in this dissertation. In the following, we highlight a number of opportunities for future research directions.

1. ***Prioritization for a wide range of vulnerabilities***. Our code artifact prioritization was primarily designed to detect buffer overflow type vulnerabilities, but the idea can be generalized to any class of bugs. Compiler-level static analysis, as used in DOWSER, will be key to ensure good precision for the prioritization early on. Another research area is to identify possibilities for prioritization within binary only programs. Existing reverse engineering techniques can give a surprisingly in-depth look into binaries today, even if they lack the precision and detail of source-based solutions. Research into this area would allow guided fuzzing to be expanded outside of the realm of software developers into the hands of third-party security analysts.

2. ***Measurable control-flow integrity for C***. Our white-box testing approach for

vtable-based call-sites was successful due to the detailed semantics encoded into these artifacts within the C++ language. Unfortunately these semantics are missing from C-style indirect call-sites, making them difficult to protect in a measurable manner. We believe however, that lessons from white-box testing might still offer an avenue to generate a benchmark with solid ground-truth for this problem space. While optimal solutions are not expected any time soon (due to the complexity of the problem at hand), having a proper standardized baseline benchmark is still preferable for future research and evaluation. A important research question on this topic is the potential to build a generic enough benchmark, to avoid researchers trying to game the benchmark itself.

3. ***Efficient memory safety for C/C++***. Finally, our metadata tracking mechanism offers a solid starting point for memory safety instrumentation, but researchers now have to look at the facilities it offers and the opportunities opened up by its characteristics. We hope that new low-overhead techniques will be soon showcased on top of METALLOC, bringing aspects of memory safety into production C/C++ systems. Such instrumentation becomes more and more valuable as data-only attacks start to proliferate and the other mitigation steps become less and less effective going forward.

References

[1] Itanium C++ ABI. `mentorembedded.github.io/cxx-abi/abi.html`.

[2] CVE-2009-2629: Buffer underflow vulnerability in nginx. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2629.

[3] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-Flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, 2005.

[4] Amir D. Aczel and Jayavel Sounderpandian. *Complete Business Statistics*. Sixth edition, 2006.

[5] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX Conference on Security*, 2010.

[6] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 263–277, 2008.

[7] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Conference on Security*, pages 51–66, 2009.

[8] S. Andersen and V. Abella. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies, Data Execution Prevention. `http://technet.microsoft.com/en-us/library/bb457155.aspx`.

[9] Aravind Prakash, Xunchao Hu, and Heng Yin. vfGuard: Strict Protection for Virtual Function Calls in COTS C++ Binaries. In *Proceedings of the Network*

and Distributed System Security Symposium, 2015.

[10] A. Arya. Analyzing chrome crash reports at scale. In *Nullcon International Security Conference Goa*, 2015.

[11] B. Aydin, G. Pataki, H. Wang, E. Bullit, and J.S. Marron. a principal component analysis for trees. *Annals of Statistics*, 3(4):1597–1615, 2009.

[12] Domagoj Babić, Lorenzo Martignoni, Stephen McCamant, and Dawn Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 12–22, 2011.

[13] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 binary executables. In *Proceedings of the International Conference on Compiler Construction*, pages 5–23, 2004.

[14] Gogul Balakrishnan, Radu Gruian, Thomas Reps, and Tim Teitelbaum. Codesurfer/x86—a platform for analyzing x86 executuables. In *Lecture Notes in Computer Science*, pages 250–254, 2005.

[15] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. SNOOZE: toward a stateful network protocol fuzZEr. In *Proceedings of the 9th International Conference on Information Security*, pages 343–358, 2006.

[16] Tao Bao, Yunhui Zheng, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Strict control dependence and its effect on dynamic information flow analyses. In *Proceedings of the 2010 International Symposium on Software Testing and Analysis*, pages 13–24, 2010.

[17] Igor Bogudlov, Tal Lev-Ami, Thomas Reps, and Mooly Sagiv. Revamping TVLA: making parametric shape analysis competitive. In *Proceedings of the International Conference on Computer Aided Verification*, pages 221–225, 2007.

[18] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: Whitebox fuzz testing in production. In *Proceedings of the International Conference on Software Engineering*, pages 122–131, 2013.

[19] Dimitar Bounov, Rami Gökhan Kıcı, and Sorin Lerner. Protecting c++ dynamic dispatch through vtable interleaving. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[20] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[21] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 209–224, 2008.

[22] Nicholas Carlini and David Wagner. ROP is Still Dangerous: Breaking Modern Defenses. In *Proceedings of the 23rd USENIX Conference on Security*, pages 385–399, 2014.

[23] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *Proceedings of the 24th USENIX Conference on Security*, pages 161–176, 2015.

[24] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. On the Limits of Information Flow Techniques for Malware Analysis and Containment. In *Proceedings of the Fifth Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, pages 143–163, 2008.

[25] Chao Zhang, Chengyu Songz, Kevin Zhijie Chen, Zhaofeng Cheny, and Dawn Song. VTint: Protecting Virtual Function Tables' Integrity. In *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[26] Xi Chen, Asia Slowinska, and Herbert Bos. Detecting custom memory allocators in C binaries. Technical report, Vrije Universiteit Amstetrdam, 2013.

[27] Yueqiang Cheng, Zongwei Zhou, Miao Yu, Xuhua Ding, and Robert H. Deng. ROPecker: A Generic and Practical Approach For Defending Against ROP Attacks. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[28] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in vivo multi-path analysis of software systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–278, 2011.

[29] Clang. Clang 3.9 documentation - control flow integrity. `http://clang.llvm.org/docs/ControlFlowIntegrity.html`.

[30] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. Losing control: On the effectiveness of control-flow integrity under stack attacks. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 952–963, 2015.

[31] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. 2001.

[32] Microsoft Corporation. Secuirty development lifecycle. `https://msdn.microsoft.com/en-us/library/bb288454.aspx`.

[33] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, 1998.

[34] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, pages 255–266, 2008.

[35] Weidong Cui, Marcus Peinado, Zhilei Xu, and Ellick Chan. Tracking rootkit footprints with a practical memory analysis system. In *Proceedings of the 21st USENIX Conference on Security*, pages 601–615, 2012.

[36] CWE/SANS. CWE/SANS TOP 25 Most Dangerous Software Errors. `www.sans.org/top25-software-errors`.

[37] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas FIsher, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. MoCFI: A Framework to Mitigate ControlFlow Attacks on Smartphones. In *Proceedings of the Network and Distributed System Security Symposium*, pages 32–44, 2012.

[38] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *Proceedings of the 23rd USENIX Conference on Security*, pages 401–416, 2014.

[39] J. DeMott. The evolving art of fuzzng. DEFCON 14, `http://www.appliedsec.com/files/The_Evolving_Art_of_Fuzzing.odp`.

[40] David Dewey and Jonathon Giffin. Static detection of c++ vtable escape vulnerabilities in binary code. In *Proceedings of the Network and Distributed System Security Symposium*, 2012.

[41] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for c with very low overhead. In *Proceedings of the International Conference on Software Engineering*, pages 162–171, 2006.

[42] Dawson Engler and Madanlal Musuvathi. Static analysis versus software model checking for bug finding. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *Lecture Notes in Computer Science*, pages 191–210. 2004.

[43] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The daikon system for dy-

namic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007.

[44] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 901–913, 2015.

[45] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9:319–349, 1997.

[46] The Mozilla Foundation. Dromaeo, javascript performance testing. `https://www.webkit.org/perf/sunspider/sunspider.html`.

[47] Vijay Ganesh, Tim Leek, and Martin Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the International Conference on Software Engineering*, pages 474–484, 2009.

[48] Michael Gegick, Laurie Williams, Jason Osborne, and Mladen Vouk. Prioritizing software security fortification through code-level metrics. In *Proceedings of the 4th ACM Workshop on Quality of Protection*, pages 31–38, 2008.

[49] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc. `http://goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[50] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 1–15, 1996.

[51] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 23–33, 2011.

[52] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[53] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated Whitebox Fuzz Testing. In *Proceedings of the Network and Distributed System Security Symposium*, pages 151–166, 2008.

[54] Enes Göktaş, Elias Athanasopoulos, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 575–589, 2014.

[55] Enes Göktaş, Elias Athanasopoulos, Michalis Polychronakis, Herbert Bos, and Georgios Portokalidis. Size Does Matter: Why Using Gadget-Chain Length to Prevent Code-Reuse Attacks is Hard. In *Proceedings of the 23rd USENIX Conference on Security*, pages 417–432, 2014.

[56] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. Undermining information hiding (and what to do about it). In *Proceedings of the 25th USENIX Conference on Security*, pages 105–119, 2016.

[57] Google. Octane benchmark. `https://code.google.com/p/octane-benchmark`.

[58] Google. Sunspider benchmark. `https://www.webkit.org/perf/sunspider/sunspider.html`.

[59] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 255–265, 2006.

[60] Wang H. and J.S. Marron. Object oriented data analysis: Sets of trees. *Annals of Statistics*, 35(5):1849–1873, 2007.

[61] Istvan Haller, Asia Slowinska, and Herbert Bos. Mempick: High-level data structure detection in c/c++ binaries. In *Proceedings of the 20th Working Conference on Reverse Engineering*, pages 32–41, 2013.

[62] Istvan Haller, Asia Slowinska, Matthias Neugschwandtner, and Herbert Bos. Dowsing for overflows: A guided fuzzer to find buffer boundary violations. In *Proceedings of the 22nd USENIX Conference on Security*, pages 49–64. USENIX, 2013.

[63] Istvan Haller, Enes Göktaş, Elias Athanasopoulos, Georgios Portokalidis, and Herbert Bos. Shrinkwrap: Vtable protection without loose ends. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 341–350, 2015.

[64] Istvan Haller, Asia Slowinska, and Herbert Bos. Scalable data structure detection and classification for c/c++ binaries. *Empirical Software Engineering*, pages 1–33, 2015.

[65] Istvan Haller, Yuseok Jeon, Hui Peng, Mathias Payer, Cristiano Giuffrida, Herbert Bos, and Erik van der Kouwe. Typesan: Practical type confusion detection. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security*, pages 517–528, 2016.

[66] Istvan Haller, Erik van der Kouwe, Cristiano Giuffrida, and Herbert Bos. Metalloc: Efficient and comprehensive metadata management for software secu-

rity hardening. In *Proceedings of the 9th ACM European Workshop on System Security*, pages 5:1–5:6, 2016.

[67] Niranjan Hasabnis, Ashish Misra, and R Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144, 2012.

[68] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[69] H. Hotelling. Analysis of a complex of statistical variables into principal components. *J. Educ. Psych.*, 24, 1933.

[70] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *Proceedings of the 24th USENIX Conference on Security*, pages 177–192, 2015.

[71] Intel. Pin - A Dynamic Binary Instrumentation Tool. `http://www.pintool.org/`.

[72] Dongseok Jang, Zachary Tatlock, and Sorin Lerner. SAFEDISPATCH: Securing C++ virtual calls from memory corruption attacks. In *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[73] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *Proceedings of the 2013 International Conference on Software Engineering*, pages 672–681, 2013.

[74] Changhee Jung and Nathan Clark. DDT: design and evaluation of a dynamic program analysis for optimizing data structure usage. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 56–66, 2009.

[75] Changhee Jung, Silvius Rus, Brian P. Railing, Nathan Clark, and Santosh Pande. Brainy: Effective selection of data structures. In *Proceedings of the 32nd ACM Conference on Programming Language Design and Implementation*, pages 86–97, 2011.

[76] R. Kaksonen. A functional method for assessing protocol implementation security. Technical Report 448, VTT, 2001.

[77] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[78] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proceedings of the 9th*

International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pages 553–568, 2003.

[79] Viktor Kuncak, Patrick Lam, Karen Zee, and Martin Rinard. Modular Pluggable Analyses for Data Structure Consistency. *IEEE Transactions on Software Engineering*, 32(12):988–1005, 2006.

[80] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, pages 147–163, 2014.

[81] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the Second International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[82] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Caver source code. `https://github.com/sslab-gatech/caver`.

[83] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2015.

[84] Byoungyoung Lee, Chengyu Song, Taesoo Kim, and Wenke Lee. Type casting verification: Stopping an emerging attack vector. In *Proceedings of the 24th USENIX Conference on Security*, pages 81–96, 2015.

[85] Jonghyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[86] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the Network and Distributed System Security Symposium*, 2010.

[87] Zhiqiang Lin, Junghwan Rhee, Xiangyu Zhang, Dongyan Xu, and Xuxian Jiang. SigGraph: Brute Force Scanning of Kernel Data Structure Instances Using Graph-based Signatures. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[88] Paul Dan Marinescu and Cristian Cadar. Make test-zesti: A symbolic execution solution for improving regression testing. In *Proceedings of the International Conference on Software Engineering*, pages 716–726, 2012.

[89] Paul Dan Marinescu and Cristian Cadar. Katch: High-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Founda-*

tions of Software Engineering, pages 235–245, 2013.

[90] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33:32–44, 1990.

[91] Mitre. Common Vulnerabilities and Exposures (CVE). `http://cve.mitre.org/`.

[92] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th conference on USENIX security symposium*, pages 67–82, 2009.

[93] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 231–245, 2007.

[94] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F Sweeney. Producing wrong data without doing anything obviously wrong! *ACM Sigplan Notices*, 44(3):265–276, 2009.

[95] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the International Conference on Software Engineering*, pages 452–461, 2006.

[96] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavy-weight Dynamic Binary Instrumentation. In *Proceedings of the Third International ACM Conference on Virtual Execution Environments*, pages 89–100, 2007.

[97] James Newsome and Dawn Song. Dynamic taint analysis: Automatic detection, analysis, and signature generation of exploit attacks on commodity software. In *Proceedings of the Network and Distributed Systems Security Symposium*, 2005.

[98] Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs. In *Proceedings of the 6th International Workshop on Security Measurements and Metrics*, pages 3:1–3:8, 2010.

[99] Ben Niu and Gang Tan. Modular Control-flow Integrity. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 577–587, 2014.

[100] Angelos Oikonomopoulos, Elias Athanasopoulos, Herbert Bos, and Cristiano Giuffrida. Poking holes in information hiding. In *Proceedings of the 25th USENIX Conference on Security*, pages 121–138, 2016.

[101] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49): 365, 1996.

[102] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Trans-

parent ROP Exploit Mitigation Using Indirect Branch Tracing. In *Proceedings of the 22nd USENIX Conference on Security*, pages 447–462, 2013.

[103] PaX Project. Address Space Layout Randomization. `http://pax.grsecurity.net/docs/aslr.txt`.

[104] Marina Polishchuk, Ben Liblit, and Chloë W. Schulze. Dynamic heap type inference for program understanding and debugging. In *Proceedings of the 34th ACM Symposium on Principles of Programming Languages*, pages 3–46, 2007.

[105] Google Chromium Project. Undefined behavior sanitizer. `https://www.chromium.org/developers/testing/undefinedbehaviorsanitizer`.

[106] Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete execution for testing nasa software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 15–26, 2008.

[107] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. Precise garbage collection for c. In *Proceedings of the International Symposium on Memory Management*, pages 39–48, 2009.

[108] Ganesan Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 119–132, 1999.

[109] Easwaran Raman and David I. August. Recursive data structure profiling. In *Proceedings of the Workshop on Memory System Performance*, pages 5–14, 2005.

[110] Thomas Reps and Gogul Balakrishnan. Improved memory-access analysis for x86 executables. In *Proceedings of the Joint European Conferences on Theory and Practice of Software*, pages 16–35, 2008.

[111] Christian Rossow, Dennis Andriesse, Tillmann Werner, Brett Stone-Gross, Daniel Plohmann, Christian J. Dietrich, and Herbert Bos. P2PWNED: Modeling and Evaluating the Resilience of Peer-to-Peer Botnets . In *Proceedings of IEEE Symposium on Security and Privacy* , pages 97–111, 2013.

[112] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[113] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit Object-oriented Programming:

On the Difficulty of Preventing Code Reuse Attacks in C++ Applications. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 745–762, 2015.

[114] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM International Symposium on Foundations of Software Engineering*, pages 263–272, 2005.

[115] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference*, page 28, 2012.

[116] Hovav Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pages 552–561, 2007.

[117] Yonghee Shin and Laurie Williams. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems*, pages 1–7, 2011.

[118] Asia Slowinska and Herbert Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *Proceedings of the ACM European Conference on Computer Systems*, pages 61–74, 2009.

[119] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: a dynamic excavator for reverse engineering data structures. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[120] Asia Slowinska, Traian Stancescu, and Herbert Bos. Body Armor for Binaries: preventing buffer overflows without recompilation. In *Proceedings of the USENIX Annual Technical Conference*, pages 125–137, 2012.

[121] Kevin Z. Snow, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, Fabian Monrose, and Ahmad-Reza Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 574–588, 2013.

[122] Alexander Sotirov. Modern exploitation and memory protection bypasses. USENIX Security invited talk, `http://www.usenix.org/events/sec09/tech/slides/sotirov.pdf`.

[123] Spike. `http://www.immunitysec.com/resources-freesoftware.shtml`.

[124] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[125] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[126] Miklos Szeredi. File system in user space. `http://fuse.sourceforge.net`.

[127] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-edge Control-flow Integrity in GCC and LLVM. In *Proceedings of the 23rd USENIX Conference on Security*, pages 340–353, 2014.

[128] Victor van der Veen, Nitish Dutt-Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory Errors: The Past, the Present, and the Future. In *Proceedings of The 15th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 86–106, 2012.

[129] Victor van der Veen, Dennis Andriesse, Enes Göktaş, Ben Gras, Lionel Sambuc, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida. Practical Context-Sensitive CFI. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security*, pages 927–940, 2015.

[130] Victor van der Veen, Enes Göktaş, Moritz Contag, Andre Pawlowski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 934–953, 2016.

[131] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. High system-code security with low overhead. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 866–879, 2015.

[132] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A Checksum-Aware Directed Fuzzing Tool for Automatic Software Vulnerability Detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 497–512, 2010.

[133] David H. White and Gerald Lüttgen. Identifying dynamic data structures by learning evolving patterns in memory. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 354–369, 2013.

[134] Nicky Williams, Bruno Marre, and Patricia Mouy. On-the-Fly Generation of K-Path Tests for C Functions. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pages 290–293, 2004.

[135] Christopher J. Van Wyk. *Data Structures and C Programs, 2nd Ed. (Addison-Wesley Series in Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1991.

[136] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In Aarti Gupta and Sharad Malik, editors, *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398, 2008.

[137] Yves Younan. FreeSentry: Protecting Against Use-After-Free Vulnerabilities Due to Dangling Pointers. In *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[138] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 349–361, 2008.

[139] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, László Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 559–573, 2013.

[140] Chao Zhang, Scott A. Carr, Tongxin Li, Yu Ding, Chengyu Song, Mathias Payer, and Dawn Song. Vtrust: Regaining trust on virtual calls. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[141] Mingwei Zhang and R. Sekar. Control flow integrity for cots binaries. In *Proceedings of the 22Nd USENIX Conference on Security*, pages 337–352, 2013.

[142] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 421–428, 2010.

[143] Misha Zitser, Richard Lippmann, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *Proceedings of the 12th ACM International Symposium on Foundations of Software Engineering*, pages 97–106, 2004.

Summary

With new security incidents surfacing almost every single day, software vendors must meet ever-increasing standards of quality in systems security. As such, major vendors now spend massive amounts of resources to test their products and minimize the probability of exploitable software bugs—the kind of bugs that allow attackers to completely compromise a target system. In particular, much effort is devoted to very dangerous bugs that corrupt memory. Despite all these efforts, memory corruption bugs are still pervasive in systems software, with well-known examples such as the Heartbleed and StageFright vulnerabilities putting millions of users at risk. Since it is impossible to completely eradicate these bugs, the only option is to mitigate their impact in practice.

This dissertation considers all the stages in which one may protect software against attacks based on memory corruption bugs: testing before deployment, detection of corruption attempts after deployment for the bugs that we cannot find by testing, and exploit containment for the bugs that we cannot easily detect. Such a symbiosis between software testing and systems security can get us closer to a solution. Lessons learned in systems security help us concentrate the testing efforts on the areas of the program which are expected to be most vulnerable. Lessons learned in software testing help us incorporate validation techniques into the workflow of security solutions. Drawing from such lessons, this dissertation presents a concert of techniques at different layers of systems security, including bug finding, memory corruption detection, and control-flow enforcement.

In bug finding, this dissertation introduces guided fuzzing based on security prioritization, which aims to prioritize the testing effort around likely vulnerable code artefacts. Such prioritization typically occurs as part of white-box testing with manual input from developers, but we show that we can automate this process using

static source analysis. This strategy lays the ground for effective modular testing, by automatically inferring the need to test certain code artefacts and avoid others.

In addition, this dissertation explores reverse engineering techniques to extend the guided fuzzing approach to scenarios where no source information is available. There is a significant gap in the literature regarding the classification of pointer-based data structures in binary programs. I bridge this gap via a novel approach based on behavioral tracking and a classifier based on refinement.

The dissertation also identifies issues in existing state-of-the-art control-flow enforcement schemes, including the one implemented in the GCC compiler today. I counter this threat by developing a novel white-box testing framework for a certain class of control-flow enforcement schemes, which allows developers to check if their mitigation fully respects C++ semantics, while at the same time offering the strongest possible protection for the program. I then leverage the framework to build a new mitigation, which fully achieves these goals, while also being faster than the variant used in GCC.

Finally, this dissertation shows that a major reason that enforcement of memory safety rules in C/C++ is inefficient lies in the overhead imposed by metadata tracking. I developed a new scheme, called variable metadata shadowing, which alleviates these concerns. My scheme is flexible enough to implement a wide range of memory safety checks, while imposing minimal performance and memory overhead during the execution. I then validate the metadata tracker in a novel type enforcement system, which detects type confusion bugs in real-world software (such as browser with over a million lines of code). The metadata tracker forms the backbone of this mitigation mechanism and I show that it can protect real applications with much greater efficiency than any other solution.

In conclusion, this dissertation presents evidence that we need a strong coupling between the areas of software testing and systems security. The best solution is to layer different techniques on top of one another and I show how each layer can be greatly enhanced via the proposed symbiosis.