Improving systems software security through program analysis and instrumentation

THÈSE N^{O} 7055 (2016)

PRÉSENTÉE LE 27 MAI 2016

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS

LABORATOIRE DES SYSTEMES FIABLES

PROGRAMME DOCTORAL EN INFORMATIQUE ET COMMUNICATIONS

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Volodymyr KUZNETSOV

acceptée sur proposition du jury:

Prof. C. Koch, président du jury
Prof. G. Candea, directeur de thèse
Prof. A.-R. Sadeghi, rapporteur
Prof. D. Boneh, rapporteur
Prof. E. Bugnion, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2016

Abstract

Security and reliability bugs are prevalent in systems software. Systems code is often written in low-level languages like C/C++, which offer many benefits but also delegate memory management and type safety to programmers. This invites bugs that cause crashes or can be exploited by attackers to take control of the program. This thesis presents techniques to detect and fix security and reliability issues in systems software without burdening the software developers.

First, we present code-pointer integrity (CPI), a technique that combines static analysis with compile-time instrumentation to *guarantee the integrity of all code pointers* in a program and thereby prevent all control-flow hijack attacks. We also present code-pointer separation (CPS), a relaxation of CPI with better performance properties. CPI and CPS offer substantially better security-to-overhead ratios than the state of the art in control flow hijack defense mechanisms, they are practical (we protect a complete FreeBSD system and over 100 packages like apache and postgresql), effective (prevent all attacks in the RIPE benchmark), and efficient: on SPEC CPU2006, CPS averages 1.2% overhead for C and 1.9% for C/C++, while CPI's overhead is 2.9% for C and 8.4% for C/C++.

Second, we present DDT, a tool for testing closed-source device drivers to automatically find bugs like memory errors or race conditions. DDT showcases a combination of a form of program analysis called selective symbolic execution with virtualization to thoroughly exercise tested drivers and produce detailed, executable traces for every path that leads to a failure. We applied DDT to several closed-source Microsoft-certified Windows device drivers and discovered 14 serious new bugs that can cause crashes or compromise security of the entire system.

Third, we present a technique for increasing the scalability of symbolic execution by merging states obtained on different execution paths. State merging reduces the number of states to analyze, but the merged states can be more complex and harder to analyze than their individual components. We introduce *query count estimation*, a technique to reason about the analysis time of merged states and decide which states to merge in order to achieve optimal net performance of symbolic execution. We also introduce *dynamic state merging*, a technique for merging states that interacts favorably with search strategies employed by practical bug finding tools, such as DDT and KLEE. Experiments on the 96 GNU Coreutils show that our approach consistently achieves several orders of magnitude speedup over previously published results.

Keywords: code-pointer integrity, control-flow integrity, program hardening, symbolic program analysis, selective symbolic execution, state merging, device driver testing.

Résumé

Les bugs de sécurité et de fiabilité sont courants dans les logiciels systèmes. Ces logiciels sont souvent écrits en langages de bas niveau comme C/C++, lesquels offrent de nombreux avantages, mais délèguent également la gestion de la mémoire et du typage aux programmeurs. Cela favorise les bugs qui causent des plantages ou qui peuvent être exploités par des attaquants afin de prendre le contrôle du programme. Cette thèse présente des techniques pour détecter et corriger les défauts de sécurité et de fiabilité dans les logiciels systèmes sans surcharger les développeurs.

Tout d'abord, nous présentons l'intégrité des pointeurs de code (IPC), une technique qui combine analyse statique et instrumentation du code à la compilation afin de *garantir l'intégrité de tous les pointeurs de code* dans un programme et ainsi empêcher toutes les attaques de détournement de flux de contrôle. Nous présentons également la séparation des pointers de code (CPS), une relaxation de l'IPC avec de meilleures propriétés de performance. IPC et CPS offrent un rapport sécurité-performance nettement meilleur que l'état de l'art dans le domaine de la défense contre le détournement du contrôle de flux, ils sont pratiques (nous protégeons un système FreeBSD complet et plus de 100 paquets comme apache et postgresql), efficaces (évitent toutes les attaques du benchmark RIPE), et performants : sur le benchamrk SPEC CPU2006, CPS a un coût moyen de 1.2% pour les programmes C et de 1.9% pour les programmes C/C++, tandis que l'IPC a un coût de 2.9% pour les programmes C et de 8.4% pour ceux en C/C++.

Deuxièmement, nous présentons DDT - un outil de test des pilotes de périphériques binaires pour la recherche automatique de bugs tels que les erreurs de mémoire ou les conditions de concurrence. DDT combine la virtualisation avec une forme d'analyse de programmes, appelée exécution symbolique sélective, afin d'exercer les pilotes de périphériques et de produire des traces exécutables détaillés pour chaque chemin menant à une erreur. Nous avons utilisé DDT sur plusieurs pilotes de périphériques Windows à source fermé certifiés par Microsoft et découvert 14 nouveaux bugs graves qui peuvent causer des plantages ou compromettre la sécurité de l'ensemble du système.

Troisièmement, nous présentons une technique qui augmente considérablement la performance de l'exécution symbolique en fusionnant les états obtenus sur différents chemins d'exécution. La fusion d'états réduit le nombre d'états à analyser, mais les états fusionnés peuvent être plus complexes et difficiles à analyser que lorsque pris séparément. Nous présentons une technique *d'estimation du nombre de requêtes*, laquelle permet d'estimer le temps

d'analyse des états fusionnés et de décider quels états fusionner afin d'obtenir un gain net de performance. Nous présentons aussi *la fusion d'état dynamique*, une technique de fusion d'états qui interagit favorablement avec les stratégies de recherche, la génération automatique de scénarios de tests et les outils de recherche de bugs, comme KLEE et DDT. Les expériences sur les 96 programmes GNU coreutils montrent que notre approche procure un gain de vitesse de plusieurs ordres de grandeur sur les résultats publiés antérieurement.

Mots clés : intégrité des pointeurs de code, intégrité du contrôle de flux, durcissement de programmes, analyse symbolique de programmes, exécution symbolique sélective, fusion d'états, analyse statique, test de pilotes de périphériques.

Acknowledgements

This thesis wouldn't have been possible without support of many great people around me.

First of all, I would like to thank George Candea, who completely redefined my expectations of what an ideal advisor should be. He showed to me, by his own example, what it takes to be a great scientist, a teacher and a leader. He taught me a lot throughout these years, and he opened many great opportunities for me to learn from others too. George always supported me throughout both ups and downs, far beyond what I expected. He set a very high bar for me that I will always strive to live up to myself, and to pass it on to the students I work with.

I would like to thank Dan Boneh, Edouard Bugnion, Christoph Koch and Ahmad-Reza Sadeghi for being on my thesis committee, for their thorough feedback on the thesis draft and the insightful questions they asked during my defense.

I would like to thank Dawn Song, who introduced me to the systems security research, for all advice and support she gave me during my internship at her lab and our subsequent collaboration.

I thank my co-authors: Stefan Bucur, George Candea, Vitaly Chipounov, Johannes Kinder, Mathias Payer, R Sekar, Dawn Song, László Szekeres, Jonas Wagner. It was an honor to work with them, and I learned a lot from each of them. I would like to thank all reviewers who provided feedback on my papers, anonymous or not, for helping to make the papers better.

I would like to thank every member of the Dependable Systems Lab for making this lab the perfect place do to research by being so inspiring, close-knit and scientifically strong team. I would like to thank Silviu Andrica, Radu Banabic, Alexandre Bique, Stefan Bucur, Amer Chamseddine, Vitaly Chipounov, Alexandru Copot, Francesco Fucci, Loïc Gardiol, Horatiu Jula, Baris Kasikci, Johannes Kinder, Lucian Mogosanu, Olivier Saudan, Georg Schmid, Benjamin Schubert, Ana Sima, Jonas Wagner, Cristian Zamfir, Peter Zankov, Arseniy Zaostrovnykh, Lisa Zhou. I am especially grateful to Nicoletta Isaac for taking care of all administrative issues arising throughout my PhD.

I would like to thank Microsoft and European Research Council for supporting my research.

I would like to thank my family: Valentyna, Olga, Katerina and Vitaliy for their continuous support and for inspiring me to start doing research in the first place. Last but not least, I would like to thank my loving wife Aliona for always supporting me throughout my PhD journey and beyond, and for making my life to much better and happier.

Contents

List of Figures

List of Tables

1 Introduction

In this chapter, we present the problem statement of this thesis (§1.1) and overview our proposed solution (§1.2).

1.1 Problem Statement

Systems code, such as OS kernels, device drivers, networking software, or language runtimes, is often written in memory-unsafe and type-unsafe languages; this makes it prone to memory, type or concurrency errors that are the primary cause of reliability issues, as well as the primary attack vector used to compromise systems security. Such errors result in material [8, 109] and human life losses [88], or serious security incidents [48]. Attackers exploit bugs, such as buffer overflows, use after free, or format string errors to cause specific corruptions of the program state that enable them to steal sensitive data or execute code that gives them control over a remote system [117, 104, 26, 14].

Security and reliability of systems software can be improved through two complimentary approaches: proactively find and fix program bugs, or mitigate their consequences when they happen. Practical mitigation techniques target specific types of bugs or specific attack or failure scenarios and cannot completely remove the need to find and fix the bugs and, hence, are complementary to bug finding tools. However, even though bug finding tools can find bugs with little or no human help, practical scalability and completeness issues result in certain bugs being missed. Furthermore, even when bugs are found, it often takes months for developers to fix them [76]. Because of such missed on long unfixed bugs, bug finding tools cannot be self-sufficient but are rather complementary to mitigation techniques.

The goal of this thesis is to strengthen the security and reliability of systems software through a combination of bug finding and mitigation techniques, without affecting the system performance or the productivity of software developers. We specifically focus on two subproblems: defeating control-flow hijack attacks, which are the most prevalent way to compromise systems software today (§1.1.1), and empowering end-users to test binary device drivers, which

run at a highest privilege level in today's OSes and yet constitute the least reliable parts of an OS kernel (§1.1.2). We also address the scalability of bug finding tools like ours and ways to improve it (§1.1.3).

1.1.1 Defeating Control-Flow Hijack Attacks on Systems Software

Control-flow hijack attacks are the most prevalent way to compromise systems software. Such attacks exploit memory corruption errors to overwrite a code pointer in program memory, such that, when that code pointer is used as a target of an indirect control flow transfer, the control flow of the program is hijacked and redirected to the location of attacker's choice. It enables the attacker to execute arbitrary computations with the privilege level of the compromised program and, in case of systems software, often gives the attacker control of the entire system.

There exist a few protection mechanisms that can reduce the risk of control-flow hijack attacks without imposing undue overheads. Data Execution Prevention (DEP) [125] uses memory page protection to prevent the introduction of new executable code into a running application. Unfortunately, DEP is defeated by code reuse attacks, such as return-to-libc [104] and return oriented programming (ROP) [117, 14], which can construct arbitrary Turing-complete computations by chaining together existing code fragments of the original application. Address Space Layout Randomization (ASLR) [123] places code and data segments at random addresses, making it harder for attackers to reuse existing code for execution. Alas, ASLR is defeated by pointer leaks, side channels attacks [68], and just-in-time code reuse attacks [118]. Finally, stack cookies [37] protect return addresses on the stack, but only against continuous buffer overflows.

Many defenses can improve upon these shortcomings but have not seen wide adoption because of the overheads they impose. According to a recent survey [122], these solutions are incomplete and bypassable via sophisticated attacks and/or require source code modifications and/or incur high performance overhead. These approaches typically employ language modifications [73, 103], compiler modifications [36, 3, 44, 101, 115], or rewrite machine code binaries [105, 137, 135]. Control-flow integrity protection (CFI) [1, 89, 135, 137, 106], a widely studied technique for practical protection against control-flow hijack attacks, was recently demonstrated to be ineffective [21, 61, 41, 22].

Memory-safe languages guarantee that a memory object can only be accessed using pointers properly based on that specific object, which in turn makes control-flow hijacks impossible, but this approach requires runtime checks to verify the temporal and spatial correctness of pointer computations, which inevitably induces undue overhead, especially when retrofitted to memory-unsafe languages. For example, state-of-the-art memory safety implementations for C/C++ incur $\geq 2\times$ overhead [102].

Existing techniques cannot both *guarantee protection* against control-flow hijacks and impose *low overhead* and *no changes* to how the programmer writes code.

1.1.2 Empowering End-Users to Test Closed-Source Device Drivers

Device drivers are one of the least secure parts of an OS kernel. Drivers and other extensions—which comprise, for instance, 70% of the Linux operating system—have a reported error rate that is 3-7 times higher than the rest of the kernel code [33], making them substantially more prone to failures or security vulnerabilities. Not surprisingly, 85% of Windows crashes are caused by driver failures [107]. Device drivers by necessity deal with untrusted data from devices or user-space applications, which opens ways for attackers to exploit driver bugs that cause such crashes to take control of the entire kernel [18].

It is therefore ironic that most computer users place full trust in closed-source binary device drivers: they run drivers (software that is often outsourced by hardware vendors to offshore programmers) inside the kernel at the highest privilege levels, yet enjoy a false sense of safety by purchasing anti-virus software and personal firewalls. Device driver flaws are more dangerous than application vulnerabilities, because device drivers can subvert the entire system and, by having direct memory access, can be used to overwrite both kernel and application memory. Recently, a zero-day vulnerability within a driver shipped with all versions of Windows allowed non-privileged users to elevate their privileges to Local System, leading to complete system compromise [97].

Our goal is to empower users to thoroughly test drivers before installing and loading them. We wish that the Windows pop-up requesting confirmation to install an uncertified driver also offered a "Test Now" button. By clicking that button, the user would launch a thorough test of the driver's binary; this could run locally or be automatically shipped to a trusted Internet service to perform the testing on behalf of the user. Such functionality would benefit not only end users, but also the IT staff charged with managing corporate networks, desktops, and servers using proprietary device drivers.

We aim to enable users to test all drivers, including those for which source code is not available, thus complementing the existing body of driver reliability techniques. There exist several tools and techniques that can be used to build more reliable drivers [50, 96, 10] or to protect the kernel from misbehaving drivers [120], but these are primarily aimed at developers who have the driver's source code. Therefore, these techniques cannot be used (or even adapted) for the use of consumers on closed-source binary drivers. Our goal is to fill this gap.

1.1.3 Improving the Scalability of Bug Finding Tools

Recent tools [57, 20, 19, 59, 31] have applied program analysis technique called symbolic execution to bug finding and automated test case generation with impressive results—they demonstrate that symbolic execution brings unique practical advantages. First, such tools perform dynamic analysis, in that they actually execute a target program and can directly execute any calls to external libraries or the operating system by concretizing arguments; this broadens their applicability to many real-world programs. Second, these tools share with static analysis

the ability to simultaneously reason about multiple program behaviors, which improves the degree of completeness they achieve. Third, symbolic execution does not use abstraction but is fully precise with respect to predicate transformer semantics [45]; it generates "per-path verification conditions" whose satisfiability implies the reachability of a particular statement, so it generally does not have false positives. Fourth, recent advances in SAT and SMT (SAT Modulo Theory) solving [49, 56, 42] have made tools based on symbolic execution significantly faster. Overall, symbolic execution promises to help solve many important, practical program analysis problems.

Nevertheless, today's symbolic execution engines still struggle to achieve scalability, because of path explosion: the number of possible paths in a program is generally exponential in its size. States in symbolic execution encode the history of branch decisions (the *path condition*) and precisely characterize the value of each variable in terms of input values (the *symbolic store*), so path explosion becomes synonymous with state explosion. Alas, the benefit of not having false positives in bug finding (save for over-approximate environment assumptions) comes at the cost of having to analyze an exponential number of states.

Our goal is to improve the scalability of bug finding tools that rely on symbolic execution by alleviating the path explosion problem.

1.2 Solution Overview

In this section, we overview the solution to the problem we described in §1.1. First, we present a program hardening technique that combines static analysis with light-weight compile-time instrumentation to prevent all control-flow hijack attacks caused by memory errors (§1.2.1). Then, we present a tool that employs analysis of program binaries to enable both developers and end-users to automatically discover reliability bugs and security vulnerabilities in closed-source binary device drivers (§1.2.2). Finally, we present a way to improve the scalability of the underlying program analysis technique employed by this tool by several orders of magnitude (§1.2.3).

1.2.1 Defeating Control-Flow Hijacks with Code-Pointer Integrity Protection

We introduce code-pointer integrity (CPI), a program hardening technique that prevents all control-flow hijack attacks, while imposing low performance overhead and requiring no changes to how programmers write code. We observe that, in order to render control-flow hijacks impossible, it is sufficient to guarantee the integrity of code pointers, i.e., those that are used to determine the targets of indirect control-flow transfers (indirect calls, indirect jumps, or returns).

The key idea is to split process memory into a *safe region* and a *regular region*. CPI uses static analysis to identify the set of memory objects that must be protected in order to guarantee

memory safety for code pointers. This set includes all memory objects that contain code pointers and all data pointers used to access code pointers indirectly. All objects in the set are then stored in the safe region, and the region is isolated from the rest of the address space (e.g., via hardware protection). The safe region can only be accessed via memory operations that are proven at compile time to be safe or that are safety-checked at runtime. The regular region is just like normal process memory: it can be accessed without runtime checks and, thus, with no overhead. In typical programs, the accesses to the safe region represent only a small fraction of all memory accesses (6.5% of all pointer operations in SPEC CPU2006 need protection). Existing memory safety techniques cannot efficiently protect only a subset of memory objects in a program, rather they require instrumenting *all* potentially dangerous pointer operations.

CPI fully protects the program against all control-flow hijack attacks that exploit program memory bugs. CPI requires no changes to how programmers write code, since it automatically instruments pointer accesses at compile time. CPI achieves low overhead by selectively instrumenting only those pointer accesses that are necessary and sufficient to formally guarantee the integrity of all code pointers. The CPI approach can also be used for data, e.g., to selectively protect sensitive information like the process UIDs in a kernel.

We also introduce code-pointer separation (CPS), a relaxed variant of CPI that is better suited for code with abundant virtual function pointers. In CPS, all code pointers and virtual table pointers are placed in the safe region, but pointers used to access code pointers or virtual table pointers indirectly are left in the regular region (such as pointers to C++ objects that contain virtual functions). Unlike CPI, CPS may allow certain control-flow hijack attacks, but it still offers stronger guarantees than CFI and incurs negligible overhead.

Our experimental evaluation shows that our proposed approach imposes sufficiently low overhead to be deployable in production. For example, CPS incurs an average overhead of 1.2% on the C programs in SPEC CPU2006 and 1.9% for all C/C++ programs. CPI incurs on average 2.9% overhead for the C programs and 8.4% across all C/C++ SPEC CPU2006 programs. CPI and CPS are effective: they prevent 100% of the attacks in the RIPE benchmark and the recent attacks [61, 41, 22] that bypass CFI, ASLR, DEP, and all other Microsoft Windows protections. We compile and run with CPI/CPS a complete FreeBSD distribution along with ≥ 100 widely used packages, demonstrating that the approach is practical.

We released our implementation of CPI open source [38], and some parts of it are already incorporated in the Clang compiler [111].

1.2.2 Automated Device Driver Testing with Selective Symbolic Execution

We present DDT, a device driver testing system that empowers end-users to test closed-source device drivers. DDT uses selective symbolic execution to explore the device driver's execution paths, and checks whether these paths can cause undesired behavior, such as crashing the

kernel or overflowing a buffer. For each suspected case of bad behavior, DDT produces a replayable trace of the execution that led to the bug, providing the consumer irrefutable evidence of the problem. The trace can be re-executed on its own, or inside a debugger.

DDT currently works for Windows device drivers. We applied it to six popular binary drivers, finding 14 bugs with relatively little effort. These include race conditions, memory bugs, use of unchecked parameters, and resource leaks, all leading to kernel crashes or hangs with potential security implications. Since DDT found bugs in drivers that have successfully passed Microsoft certification, we believe it could be used to improve the driver certification process.

1.2.3 Improving Scalability of Symbolic Execution with Efficient State Merging

We present a technique to improve the scalability of bug finding tools that rely on symbolic execution, such as DDT, by alleviating the path explosion problem of symbolic execution.

One way to reduce the number of states is to merge states that correspond to different paths. This is standard in classic static analysis, where the resulting merged state over-approximates the individual states that were merged. Several techniques, such as ESP [39] and trace partitioning [92], reduce but do not eliminate the resulting imprecision (which can be a source of false positives) by associating separate abstract domain elements to some sets of execution paths. In symbolic execution, as a matter of principle, a merged state would have to precisely represent the information from *all* execution paths *without any* over-approximation. Consider, for example, the program *if (x<0) {x=0;} else {x=5;}* with input X assigned to x. We denote with (pc, s) a state that is reachable for inputs obeying path condition pc and in which the symbolic store $s = [v_0 = e_0, \ldots, v_n = e_n]$ maps variable v_i to expression e_i, respectively. In this case, the two states $(X < 0, [x = 0])$ and $(X \geq 0, [x = 5])$, which correspond to the two feasible paths, can be merged into one state $(\text{true}, [x = \text{ite}(X < 0, 0, 5)])$. Here, $\text{ite}(c, p, q)$ denotes the if-then-else operator that evaluates to p if c is true, and to q otherwise. If states were merged this way for every branch of a program, symbolic execution would become similar to verification condition generation or bounded model checking, where the entire problem instance is encoded in one monolithic formula that is passed in full to a solver.

State merging effectively decreases the number of paths that have to be explored [58, 62], but also increases the size of the symbolic expressions describing variables. Merging introduces disjunctions, which are notoriously difficult for SMT solvers, particularly for those using eager translation to SAT [56]. Merging also converts differing concrete values into symbolic expressions, as in the example above: the value of x was concrete in the two separate states, but symbolic ($\text{ite}(X < 0, 0, 5)$) in the merged state. If x were to appear in branch conditions or array indices later in the execution, the choice of merging the states may lead to more solver invocations than without merging. This combination of larger symbolic expressions and extra solver invocations can drown out the benefit of having fewer states to analyze, leading to an actual *decrease* in the overall performance of symbolic execution [62].

Furthermore, state merging conflicts with important optimizations in symbolic execution: search-based symbolic execution engines, like the ones used in test case generators and bug finding tools, employ *search strategies* to prioritize searching of "interesting" paths over "less interesting" ones, e.g., with respect to maximizing line coverage given a fixed time budget. To maximize the opportunities for state merging, however, the engine would have to traverse the control flow graph in topological order, which typically contradicts the strategy's path prioritization policy.

We present a solution to these to challenges that yields a net benefit in practice. We combine the state space reduction benefits of merged exploration with the constraint solving benefits of individual exploration, while mitigating the ensuing drawbacks. Experiments on the GNU COREUTILS show that employing our approach in a symbolic execution engine achieves speedups over the state of the art that are exponential in the size of symbolic input. Specifically, we introduce:

- *Query count estimation*, a way to statically approximate the number of times each variable will appear in future solver queries after a potential merge point. We then selectively merge two states only when we expect differing variables to appear infrequently in later solver queries. Since this selective merging merely groups paths instead of pruning them, inaccuracies in the estimation do not hurt soundness or completeness.
- *Dynamic state merging*, a merging algorithm specifically designed to interact favorably with search strategies. The algorithm explores paths independently of each other and uses a similarity metric to identify on-the-fly opportunities for merging, while preserving the search strategy's privilege of dictating exploration priorities.

2 Related Work

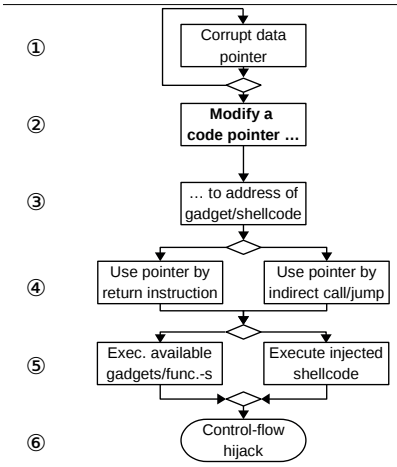
In this chapter we review related work. We first discuss program hardening techniques aimed to prevent control-flow hijack attacks (§2.1). Then we discuss testing, verification and bug finding techniques aimed to strengthen the security and reliability of device drivers (§2.2). Finally, we overview the landscape of generic program analysis techniques (§2.3), introducing the generalized symbolic execution algorithm that can be instantiated to implement different flavors of program analysis (§2.3.1) and analyzing the trade-offs such flavors make (§2.3.2 and §2.3.3).

2.1 Control-Flow Hijack Defense Mechanisms

A variety of defense mechanisms have been proposed to-date to answer the increasing challenge of control-flow hijack attacks. Figure 2.1 compares the design of the different protection approaches to our approach.

Enforcing memory safety ensures that no dangling or out-of-bounds pointers can be read or written by the application, thus preventing the attack in its first step. Cyclone [73] and CCured [103] extend C with a safe type system to enforce memory safety features. These approaches face the problem that there is a large (unported) legacy code base. In contrast, CPI and CPS both work for unmodified C/C++ code. SoftBound [101] with its CETS [102] extension enforces *complete* memory safety at the cost of $2\times - 4\times$ slowdown. Tools with less overhead, like BBC [4], only *approximate* memory safety. LBC [63] and Address Sanitizer [115] detect continuous buffer overflows and (probabilistically) indexing errors, but can be bypassed by an attacker who avoids the red zones placed around objects. Write integrity testing (WIT) [3] provides spatial memory safety by restricting pointer writes according to points-to sets obtained by an over-approximate static analysis (and is therefore limited by the static analysis). Other techniques [44, 2] enforce type-safe memory reuse to mitigate attacks that exploit temporal errors (use after frees).

CPI by design enforces spatial and temporal memory safety for a subset of data (code pointers)



Property	Mechanism	Stops all control-flow hijacks?	Avg. overhead
Memory Safety	SoftBound+CETS [34, 35]	**Yes**	116%
	BBC [4],	No: sub-objects, reads not protected	110%
	LBC [20], ASAN [43],	No: protects red zones only	23%
	WIT [3]	No: over-approximate valid sets	7%
Code-Pointer Integrity (this work)	CPI	**Yes**	8.4%
	CPS	No: valid code ptrs. interchangeable	1.9%
	Safe Stack	No: precise return protection only	~0%
Randomization	ASLR [40], ASLP [26]	No: vulnerable to information leaks	~10%
	PointGuard [13]	No: vulnerable to information leaks	10%
	DSR [6]	No: vulnerable to information leaks	20%
	NOP insertion [21]	No: vulnerable to information leaks	2%
Control-Flow Integrity	Stack cookies	No: probabilistic return protection only	~2%
	CFI [1]	No: over-approximate valid sets	20%
	WIT (CFI part) [3]	No: over-approximate valid sets	7%
	DFI [10]	No: over-approximate valid sets	104%
Non-Executable Data	HW (NX bit)	No: code reuse attacks	0%
	SW (Exec Shield, PaX)	No: code reuse attacks	few %
High-level policies	Sandboxing (SFI)	Isolation only	varies
	ACLs	Isolation only	varies
	Capabilities	Isolation only	varies

Figure 2.1 – Summary of control-flow hijack defense mechanisms aligned with individual steps that are necessary for a successful attack. The figure on the left is a simplified version of the complete memory corruption diagram in [122].

in Step 2 of Figure 2.1. Our Levee prototype currently enforces spatial memory safety and may be extended to enforce temporal memory safety as well (e.g., how CETS extends SoftBound). We believe CPI is the first to stop all control-flow hijack attacks at this step.

Randomization techniques, like ASLR [123] and ASLP [75], mitigate attacks by restricting the attacker's knowledge of the memory layout of the application in Step 3. PointGuard [36] and DSR [13] (which is similar to probabilistic WIT) randomize the data representation by encrypting pointer values, but face compatibility problems. Software diversity [67] allows fine-grained, per-instance code randomization. Randomization techniques are defeated by information leaks through, e.g., memory corruption bugs [118] or side channel attacks [68].

Control-flow integrity [1] ensures that the targets of all indirect control-flow transfers point to valid code locations in Step 4. All CFI solutions rely on statically pre-computed context-insensitive sets of valid control-flow target locations. Many practical CFI solutions simply include every function in a program in the set of valid targets [135, 137, 89, 124]. Even if precise static analysis would be feasible, CFI could not guarantee protection against all control-flow hijack attacks, but rather merely restrict the sets of potential hijack targets. Indeed, recent results [61, 41, 22] show that many existing CFI solutions can be bypassed in a principled way. CFI+SFI [134], Strato [133] and MIPS [105] enforce an even more relaxed, statically defined CFI property in order to enforce software-based fault isolation (SFI). CCFI [91] encrypts code pointers in memory and provides security guarantees close to CPS. Data-flow based techniques like data-flow integrity (DFI) [23] or dynamic taint analysis (DTA) [113] can enforce that the used code pointer was not set by an unrelated instruction or to untrusted data, respectively. These techniques may miss some attacks or cause false positives, and have higher performance costs than CPI and CPS. Stack cookies, CFI, DFI, and DTA protect control-transfer instructions

by detecting illegal modification of the code pointer whenever it is used, while CPI protects the load and store of a code pointer, thus preventing the corruption in the first place. CPI provides precise and provable security guarantees.

In Step 5, the execution of injected code is prevented by enforcing the non-executable (NX) data policy, but code-reuse attacks remain possible.

High level policies, e.g., restricting the allowed system calls of an application, limit the power of the attacker even in the presence of a successful control-flow hijack attack in Step 6. Software fault isolation (SFI) techniques [93, 51, 24, 132, 134] restrict indirect control-flow transfers and memory accesses to part of the address space, enforcing a sandbox that contains the attack. SFI prevents an attack from escaping the sandbox and allows the enforcement of a high-level policy, while CPI enforces the control-flow inside the application.

CPI and CPS rely on an instruction-level isolation mechanism to enforce the separation of code pointers from the rest of data in program memory (§3.2). This thesis presents multiple implementations of such mechanism (§3.4.3), including implementations that provide precise isolation guarantees (based on hardware- or software-enforced isolation), as well as probabilistic guarantees (based on randomization and information hiding). Evans et al. [52] demonstrated that one of the implementations with probabilistic guarantees can be bypassed in practical settings. Their attack cannot subvert the other implementation alternatives presented in this thesis (see §3.4.3 and [81]).

2.2 Strengthening Security and Reliability of Device Drivers

Two main approaches have been taken to improve the safety and reliability of device drivers. Testing and verification approaches concentrate on finding or avoiding bugs before the driver is shipped (§2.2.1). However, thoroughly testing a driver to the point of guaranteeing the absence of bugs or proving all aspects of its correctness is still economically infeasible, so bugs frequently make their way to the field. Mitigation approaches aim to protect systems from bugs that are missed during the testing phase, but typically at the expense of runtime overhead and/or modifications to the OS kernel (§2.2.2).

2.2.1 Device Driver Testing and Verification

Testing device drivers can be done statically or dynamically. For example, SLAM [10] statically checks the source code of a device driver for correct Windows API usage. It uses a form of model checking combined with an abstract representation of the source code, suitable for the properties to be checked. However, it is subject to false positives and false negatives stemming from incomplete and/or imprecise API models. Random testing, such as blackbox fuzzing [95], is dynamic and has no false positives, but has larger number of false negatives compared to symbolic execution [19].

Microsoft provides various tools for stress-testing device drivers running in their real environment. For example, Driver Verifier [98] provides deep testing of running device drivers, but it can miss rarely executed code paths. DDT combines the power of both static and dynamic tools: it runs drivers in a real environment, and combines its own checks with those of the Driver Verifier. Moreover, DDT employs fully symbolic hardware, leading to a more thorough exploration.

Selective symbolic execution was first introduced in [30] and later reused in [29]. DDT shares common ideas with these, but is also distinguished by several aspects.

First, reverse engineering of a driver with RevNIC does not require execution to be sound. For example, RevNIC overwrites with unconstrained symbolic values the concrete parameters passed by the OS to the driver. In contrast, since DDT is a testing tool, it requires the execution to be sound to avoid false positives. This introduces additional requirements on injection of symbolic values and on concretization. For example, the concrete packet size must be replaced by a symbolic value constrained not to be greater than the original value, to avoid buffer overflows.

Second, DDT introduces the use of lightweight API annotations to describe the interface between a driver and a kernel. Annotations encode developers' knowledge about a specific kernel API, and help improve code coverage as well as detect more logic bugs. Such annotations were not present in RevNIC.

Third, DDT mixes in-VM instrumentation (bug checking) with instrumentation from outside the VM. DDT can reuse existing bug-finding tools that run in the guest OS, extending these tools with symbolic execution to work on multiple paths.

Finally, during symbolic execution, RevNIC only gathers executed LLVM code and traces of device accesses. In contrast, DDT analyzes the execution in order to track the origin of symbolic values and control flow dependencies through the path leading to a bug. DDT generates annotated execution traces and input values that help developers reproduce and understand the bugs.

The idea of replacing reads from hardware with symbolic values has been mentioned before [15]. With DDT, we introduce the new concept of fully symbolic hardware, which can interact both with concretely running OSes and with symbolically running device drivers. Fully symbolic hardware can also issue symbolic interrupts, enabling the testing of various interleavings of device driver code and interrupt handlers.

S^2E [31, 32] extends the idea of selective symbolic execution to build a universal platform for in-vivo multi-path analysis of software systems. We re-implemented DDT on top of the S2E platform [31].

SymDrive [110] builds upon S^2E and the ideas behind DDT, but supports more OSes, has simpler interface to write checkers, and provides an execution tracing tool to identify how

a patch changes the execution of a driver. SymDrive also implements a source-to-source transformation aimed to make symbolic execution faster.

Recent efforts of formally verifying operating systems kernels [80, 64], file systems [27], and device drivers [64] show great promise, but still remain too expensive for widespread adoption.

2.2.2 Device Driver Failures Mitigation

When testing is not enough, it is possible to continuously monitor the drivers at runtime and provide information on the cause of the crashes. For example, Nooks [121] combines in-kernel wrapping and hardware-enforced protection domains to trap common faults and recover from them. Nooks works with existing drivers, but requires source code and incurs runtime overhead.

SFI [127] and XFI [50] use faster software isolation techniques and provide fine grained isolation of the drivers to protect the kernel from references outside their logical protection domain. However, it can only protect against memory failures and incurs runtime overhead. XFI can work on binary drivers but still requires debugging information for the binaries in order to reliably disassemble them. SafeDrive [139] uses developer provided annotations to enforce type-safety constraints and system invariants for kernel-mode drivers written in C. Finally, BGI [25] provides byte-granularity memory protection to sandbox kernel extensions. BGI was also able to find driver bugs that manifest when running the drivers with BGI isolation. However, BGI also requires access to the source code and incurs runtime overhead.

Minix [65] explicitly isolate drivers by running them in distinct address spaces; this approach is suitable for microkernels. Vino [114] introduces an alternative OS design, which combines software fault isolation with a lightweight transactional system to protect against large classes of problems in kernel extensions.

2.3 Trade-offs in Symbolic Program Analysis

Symbolic execution is just one of a multitude of precise symbolic program analyses. Tools such as CBMC [35], Saturn [130], and Calysto [9] have shown that exact, abstraction-free path sensitive local reasoning is feasible and can be fully outsourced to an external solver. With the help of a generic worklist algorithm (§2.3.1), we illustrate the relationship among precise symbolic program analyses and explain the trade-offs in the resulting solver queries (§2.3.2). We then overview approaches to achieve scalability in other types of program analysis (§2.3.3).

2.3.1 General Symbolic Exploration

Precise symbolic program analyses essentially perform forward expression substitution starting from a set of input variables. The resulting formulae are then used to falsify assertions and

find bugs, or to generate input assignments and generate test cases. Algorithm 1 is a generic algorithm for symbolic program analysis that can be used to implement different analysis flavors. For illustration purposes, we consider only a simple input language with assignments, conditional goto statements, assertions, and halt statements.

The algorithm is parameterized by a function *pickNext* for choosing the next state in the worklist, a function *follow* that returns a decision on whether to follow a branch, and a relation $\sim$ that controls whether states should be merged. We now extend the notation for states used in §1.2.3 to triples (ℓ, pc, s), consisting of a program location ℓ, the path condition pc, and the symbolic store s that maps each variable to either a concrete value or an expression over input variables. In line 1, the worklist w of the algorithm is initialized with a state whose symbolic store maps each variable to itself (for simplicity, we exclude named constants). Here, $\lambda x.e$ denotes the function mapping parameter x to an expression e (we will use $\lambda(x_1,\ldots x_n).e$ for multiple parameters). In each iteration, the algorithm picks a new state from the worklist (line 3).

On encountering an assignment $v := e$ (lines 5-6), the algorithm creates a successor state at the fall-through successor location $succ(\ell)$ of ℓ by updating the symbolic store s with a mapping from v to a new symbolic expression obtained by evaluating e in the context of s, and adds the new state to the set S. At every branch (lines 7-11), the algorithm first checks whether to follow either path and, if so, adds the corresponding condition to the successor state, which in turn is added to S. Analyses can decide to not follow a branch if the branch is infeasible or would exceed a limit on loop unrolling. For assertions (line 12-14), the path condition, the symbolic store, and the negated assertion are put in conjunction and checked for satisfiability. Since the algorithm does not over-approximate, this check has no false positives. Halt statements terminate the analyzed program, so the algorithm just outputs the path condition, a satisfying assignment of which can be used to generate a test case for the execution leading to the halt.

In lines 17-22, the new states in S are then merged with any matching states in the worklist before being added to the worklist themselves. Two states match if they share the same location and are similar according to $\sim$. Merging creates a disjunction of the two path conditions (which can be simplified by factoring out common prefixes) and builds the merged symbolic store from ite expressions that assert one or the other original value, depending on the path taken (line 20). The ite expressions that assert an identical value in both cases (because it was equal in both symbolic stores) can be simplified to that value.

2.3.2 The Design Space of Symbolic Program Analysis

The differences between various implementations of precise symbolic analysis lie in the following aspects:

1. the handling of loops and/or recursion;
2. whether and how the feasibility of individual branches is checked to avoid encoding

infeasible paths;
3. whether and how states from different paths are merged;
4. compositionality, i.e., the use of function summaries.

Loops affect soundness and completeness, while the other aspects are trade-offs that critically affect analysis performance. We now illustrate these different aspects using Algorithm 1.

Loops and Recursion. Bounded model checkers [35] and extended static checkers [53, 130, 9] unroll loops up to a certain bound, which can be iteratively increased if an injected unwinding assertion fails. Such unrolling is usually performed by statically rewriting the CFG, but can be fit into Algorithm 1 by defining *follow* to return false for branches that would unroll a loop beyond the bound. Symbolic execution explores loops as long as it cannot prove the infeasibility of the loop condition. A search strategy, implemented in the function *pickNext*, can bias the analysis against states that perform many repetitions of the same loop. For example, a search strategy optimized for line coverage selects states close to unexplored code and avoids states in deep loop unrollings [19].

Dynamic test generation as implemented in DART [57] starts with an arbitrary initial unrolling of the loop and explores different unrollings in subsequent tests. That is, DART implements *pickNext* to follow concrete executions, postponing branch alternatives until they are covered by a subsequent concrete execution.

All these approaches essentially perform loop unrolling and are generally incomplete for finite analysis times. Loop invariants are rarely used (though [60] is an exception) since weak invariants can introduce false positives, which these precise analyses are specifically designed to avoid. Weakest precondition-based program verification engines such as Boogie [87] and Havoc [83], which also interface with external solvers, rely on the user to supply sufficiently strong invariants for proving all properties of interest.

Feasibility Checking. While performing expression substitution along individual paths, certain combinations of conditional branches can turn out to be infeasible. Not propagating states that represent infeasible paths helps to reduce path explosion by investing solving time earlier in the execution. Intermediate feasibility checks are usually performed only by symbolic execution engines that follow a single path at a time (when reasoning about groups of paths, branches are less likely to be infeasible), in which case *follow* simply invokes the constraint solver.

State Merging. When states meet at the same control location, there are two general possibilities for combining their information: either the states are maintained separately, or the states are merged into a single state. In precise symbolic analysis, merging is not allowed to introduce abstraction. From a conceptual viewpoint, state merging therefore only changes the shape of a formula that characterizes a set of execution paths: if states are kept separate, a set of paths is described by their disjunction; if states are merged, there is only one formula with disjunctions in the path condition and ite expressions in the symbolic store that guard

the values of the variables depending on the path taken.

In general, we distinguish two extremes: (i) complete separation of paths, as implemented by *search-based symbolic execution* (e.g., [16, 77, 57, 20, 19, 59]), and (ii) complete *static state merging*, as implemented by verification condition generators (e.g., [35, 72, 130, 9]). Static state merging combines states at join points after completely encoding all subpaths, i.e., it defines *pickNext* to explore all subpaths leading to a join point before picking any states at the join point, and it defines $\sim$ to contain all pairs of states. In search-based symbolic execution engines, *pickNext* can be chosen freely according to the search goal, and $\sim$ is empty. Thus, they can, for example, choose to explore just the successors of a specific state and delay exploration of additional loop iterations.

Some approaches adopt intermediate merging strategies. In the context of bounded model checking (BMC), Ganai and Gupta [54] investigate splitting the verification condition along subsets of paths. This moves BMC a step into the direction of symbolic execution, and corresponds to partitioning the $\sim$ relation. Hansen et al. [62] describe an implementation of static state merging in which they modify the exploration strategy to effectively traverse the CFG in topological order and merge all states that share the same program location. For two of their three tested examples, the total solving time increases with this strategy thus showing this approach to be sub-optimal. Another prominent example of state merging is the use of function summaries in symbolic execution, which we explain below.

Compositionality. For precise interprocedural symbolic execution, the simplest and most common approach is function inlining. This causes functions to be re-analyzed at every call site, which could be avoided using function summaries. Summaries that do not introduce abstraction and are thus suitable for symbolic execution can be implemented by computing an intraprocedural path condition in terms of function inputs, and then merging all states at the function exit.

Alas, applying such a function summary is essentially as expensive as re-analyzing the function, if the translation effort from the programming logic into the representation logic is negligible. Using a summary instead of inlining avoids only the feasibility checks for intraprocedural paths that are infeasible regardless of the function input. The cost of the other feasibility checks that a non-compositional symbolic execution would perform is not eliminated by function summaries. Instead, the branch conditions are contained in the ite expressions of the summary and will increase the complexity of later SMT queries.

For dynamic test generation, Godefroid [58] suggests to collect summaries as disjunctions of pairs of input and output constraints. In further work [7], this is extended to record summaries one path at a time and to apply partial summaries whenever they match the input preconditions. Dynamic test generation re-executes the full program (with heavy instrumentation) for each branch of which the alternate case is to be analyzed. Due to re-execution, analyzing all branches in functions would come at an especially high cost, so the savings outweigh the additional solving costs for the merged summary states.

For simplicity, Algorithm 1 is just intraprocedural, supporting function calls by inlining. It can generate precise symbolic function summaries, if invoked per procedure and with a similarity relation that merges all states when the function terminates.

2.3.3 Scalability of Program Analysis Beyond Symbolic Execution

We discussed the most closely related work in §2.3.2, where we focused on what we called precise symbolic program analysis. In this section, we look a bit further into alternative approaches that are similar in that they build symbolic expressions and rely on SAT or SMT solving, but use other techniques for improving scalability.

A first class of techniques focuses on pruning redundant states in symbolic execution. Boonstoppel et al. [15] dynamically determine variable liveness during symbolic execution. Their analysis considers the already explored paths through the current statement and determines the variables that are dead on all paths. It then uses the rest of the variables to check whether a state is equivalent to a previously explored one and can be safely pruned. In a sense, this is a special case of QCE, where no merging is performed unless the differing variables never used again. In our approach, we do not actually prune paths but still represent them in the merged state. If the differing variables are never used, this is equivalent to pruning one of the paths. This allows us to use imprecise static analysis and to merge in cases where variables are not dead but just rarely used.

McMillan [94] introduces lazy annotation in symbolic execution to build summaries on the fly and generalize them by Craig interpolation. This generalization goes beyond regular symbolic summaries, but is also computationally more expensive; we would like to measure the net effectiveness of this technique in future work.

Jonas et al. [126] introduce a set of compiler transformations aimed at making subsequent analysis of the intermediate representation code or binaries generated by the compiler faster. Some of these transformations restructure the program control-flow in a way that is equivalent to state merging.

A proven effective way to scale up symbolic program analysis is to forgo precision and introduce abstraction. Saturn [130] uses a symbolic exploration algorithm to build verification conditions for specific properties. It is specifically designed to find bugs in large system software and therefore sacrifices precision at several points. Loops are unrolled just once, and functions are aggressively summarized. Similarly, Calysto [9] relies on structural abstraction to initially represent function effects as fresh variables. False positives are iteratively eliminated by replacing these variables with precise function summaries.

The bounded model checker in the Varvel/F-Soft verification platform uses lightweight static analysis to infer over-approximate function summaries that are only applied below a configurable depth in the call graph [72, 71]. Therefore, it introduces abstraction only at deeper levels, in an effort to reduce false positives. Sery et al. [116] describe the use of over-approximate sum-

maries in bounded model checking. Whenever assertion violations are found, their method falls back to inlining, to avoid false positives. Therefore, speedups are only attainable for successful verification runs.

Abstraction-based analyses could scale significantly better than symbolic execution. However, they are prone to false positives and, perhaps more importantly, are harder to deploy. Symbolic execution engines do not require hand-written stubs for external functions or system calls, but can instead simply execute the call by concretizing its parameters. This sacrifices the theoretical guarantee of eventually achieving complete path coverage, but is a significant advantage for test case generation and bug finding.

Input: Choice function *pickNext*, similarity relation $\sim$, branch checker *follow*, and initial location ℓ_0.
Data: Worklist w and set of successor states S.

```
1  w := {(ℓ0, true, λv.v)};
2  while w ≠ ∅ do
3      (ℓ, pc, s) := pickNext(w); S := ∅;
       // Symbolically execute the next instruction
4      switch instr(ℓ) do
5          case v := e                                              // assignment
6              S := {(succ(ℓ), pc, s[v ↦ eval(s, e)])};
7          case if(e) goto ℓ'                                       // conditional jump
8              if follow(pc ∧ s ∧ e) then
9                  S := {(ℓ', pc ∧ e, s)};
10             if follow(pc ∧ s ∧ ¬e) then
11                 S := S ∪ {(succ(ℓ), pc ∧ ¬e, s)};
12         case assert(e)                                           // assertion
13             if isSatisfiable(pc ∧ s ∧ ¬e) then abort;
14             ;
15             else S := {(succ(ℓ), pc, s)};
16             ;
17         case halt                                                // program halt
18             print pc;
       // Merge new states with matching ones in w
19     forall (ℓ'', pc', s') ∈ S do
20         if ∃(ℓ'', pc'', s'') ∈ w : (ℓ'', pc'', s'') ~ (ℓ'', pc', s') then
21             w := w \ {(ℓ'', pc'', s'')};
22             w := w ∪ {(ℓ'', pc' ∨ pc'', λv.ite(pc', s'[v], s''[v]))};
23         else
24             w := w ∪ {(ℓ'', pc', s')};
25 print "no errors";
```

Algorithm 1. Generic symbolic exploration.

3 Defeating Control-Flow Hijacks with Code-Pointer Integrity Protection

In this chapter, we present Code-Pointer Integrity (CPI), a protection mechanism that prevents all control-flow hijack attacks that are caused by memory corruption errors with low performance overhead. CPI instruments C/C++ programs at compile time, enforcing precise memory safety for all direct and indirect pointers to code in a program, which ensures the above security guarantee. Code-Pointer Separation (CPS) is a simplified version of CPI that provides practical protection against most control-flow hijack attacks by ensuring the integrity of direct pointers to code only. The performance overhead of our CPI implementation on SPEC2006 benchmarks is 8.4% on average, while the performance overhead of CPS is 1.9% on average.

CPI achieves low performance overhead by limiting memory safety enforcement to sensitive pointers only (i.e., direct or indirect pointers to code). The key idea is to split program memory into two isolated regions: the safe region stores all sensitive pointers, and the regular region stores everything else. CPI uses static analysis to identify program instructions that may access the safe region, and instruments them with memory safety checks. CPI employs instruction-level isolation in order to prevent all other instructions from accessing the safe region, even if hijacked by the attacker. In order to avoid changing the memory layout of the regular region, CPI reserves the locations normally occupied by sensitive pointers in the regular region and, in the safe region, it maintains a map from the addresses of these reserved locations to pointer values and corresponding metadata required for memory safety checks.

An implementation of CPI or CPS consists of (i) a static analysis pass that splits all memory accesses in a program into those that may access sensitive pointers and those that cannot, (ii) an instrumentation pass that instruments all accesses that might access sensitive pointers to use the safe region and inserts runtime checks that enforce memory safety of these accesses, and (iii) an instruction-level isolation mechanism that prevents all memory accesses that are not instrumented with memory safety checks from ever accessing the safe region, even if a memory access is compromised by an attacker.

This chapter is organized as follows: we introduce our threat model (§3.1), describe CPI

and CPS design (§3.2), present the formal model of CPI (§3.3), and our implementation of CPI (§3.4).

3.1 Threat Model

This chapter is concerned solely with control-flow hijack attacks, namely ones that give the attacker control of the instruction pointer. The purpose of this type of attack is to divert control flow to a location that would not otherwise be reachable in that same context, had the program not been compromised. Examples of such attacks include forcing a program to jump (i) to a location where the attacker injected shell code, (ii) to the start of a chain of return-oriented program fragments ("gadgets"), or (iii) to a function that performs an undesirable action in the given context, such as calling `system()` with attacker-supplied arguments. Data-only attacks, i.e., that modify or leak unprotected non-control data, are out of scope.

We assume powerful yet realistic attacker capabilities: full control over process memory, but no ability to modify the code segment. Attackers can carry out arbitrary memory reads and writes by exploiting input-controlled memory corruption errors in the program. They cannot modify the code segment, because the corresponding pages are marked read-executable and not writable, and they cannot control the program loading process. These assumptions ensure the integrity of the original program code instrumented at compile time, and enable the program loader to safely set up the isolation between the safe and regular memory regions. Our assumptions are consistent with prior work in this area.

3.2 Design

We now present the terminology used to describe our design, then define the code-pointer integrity property (§3.2.1), describe the corresponding enforcement mechanism (§3.2.2), and define a relaxed version that trades some security guarantees for performance (§3.2.4). We further formalize the CPI enforcement mechanism and sketch its correctness proof in §3.3.

We say a pointer dereference is *safe* iff the memory it accesses lies within the target object on which the dereferenced pointer is based. A *target object* can either be a memory object or a control flow destination. By *pointer dereference* we mean accessing the memory targeted by the pointer, either to read/write it (for data pointers) or to transfer control flow to its location (for code pointers). A *memory object* is a language-specific unit of memory allocation, such as a global or local variable, a dynamically allocated memory block, or a sub-object of a larger memory object (e.g., a field in a `struct`). Memory objects can also be program-specific, e.g., when using custom memory allocators. A *control flow destination* is a location in the code, such as the start of a function or a return location. A target object always has a well defined lifetime; for example, freeing an array and allocating a new one with the same address creates a different object.

We say a pointer is *based on* a target object X iff the pointer is obtained at runtime by (i) allocating X on the heap, (ii) explicitly taking the address of X, if X is allocated statically, such as a local or global variable, or is a control flow target (including return locations, whose addresses are implicitly taken and stored on the stack when calling a function), (iii) taking the address of a sub-object y of X (e.g., a field in the X struct), or (iv) computing a pointer expression (e.g., pointer arithmetic, array indexing, or simply copying a pointer) involving operands that are either themselves based on object X or are not pointers. This is slightly stricter version of C99's "based on" definition: we ensure that each pointer is based on at most one object.

The execution of a program is *memory-safe* iff all pointer dereferences in the execution are safe. A program is memory-safe iff all its possible executions (for all inputs) are memory-safe. This definition is consistent with the state of the art for C/C++, such as SoftBounds+CETS [101, 102]. Precise memory safety enforcement [101, 103, 73] tracks the based-on information for each pointer in a program, to check the safety of each pointer dereference according to the definition above; the detection of an unsafe dereference aborts the program.

3.2.1 The Code-Pointer Integrity (CPI) Property

A program execution satisfies the code-pointer integrity property iff all its dereferences that either dereference or access sensitive pointers are safe. *Sensitive pointers* are code pointers and pointers that may later be used to access sensitive pointers. Note that the sensitive pointer definition is recursive, as illustrated in Figure 3.1. According to case (iv) of the based-on definition above, dereferencing a pointer to a pointer will correspondingly propagate the based-on information; e.g., an expression `*p = &q` copies the result of `&q`, which is a pointer based on `q`, to a location pointed to by `p`, and associates the based-on metadata with that location. Hence, the integrity of the based-on metadata associated with sensitive pointers requires that pointers used to update sensitive pointers be sensitive as well (we discuss implications of relaxing this definition in §3.2.4). The notion of a sensitive pointer is dynamic. For example, a `void*` pointer 2 in Figure 3.1 is sensitive when it points at another sensitive pointer at run time, but it is not sensitive when it points to an integer.

A memory-safe program execution trivially satisfies the CPI property, but memory-safety instrumentation typically has high runtime overhead, e.g., $\geq 2\times$ in state-of-the-art implementations [102]. Our observation is that only a small subset of all pointers are responsible for making control-flow transfers, and so, by enforcing memory safety only for control-sensitive data (and thus incurring no overhead for all other data), we obtain important security guarantees while keeping the cost of enforcement low. This is analogous to the control-plane/data-plane separation in network routers and modern servers [5], with CPI ensuring the safety of data that influences, directly or indirectly, the control plane.

Determining precisely the set of pointers that are sensitive can only be done at run time. However, the CPI property can still be enforced using any over-approximation of this set, and

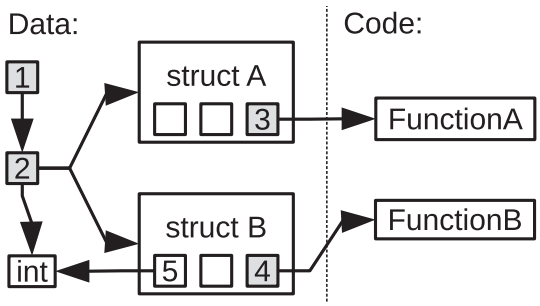


Figure 3.1 – CPI protects code pointers 3 and 4 and pointers 1 and 2 (which may access pointers 3 and 4 indirectly). Pointer 2 of type void* may point to different objects at different times. The int* pointer 5 and non-pointer data locations are not protected.

such over-approximations can be obtained at compile time, using static analysis.

3.2.2 The CPI Enforcement Mechanism

We now describe a way to retrofit the CPI property into a program *P* using a combination of static instrumentation and runtime support. Our approach consists of a *static analysis* pass that identifies all sensitive pointers in *P* and all instructions that operate on them (§3.2.2), an *instrumentation* pass that rewrites *P* to "protect" all sensitive pointers, i.e., store them in a separate, safe memory region and associate, propagate, and check their based-on metadata (§3.2.2), and an instruction-level *isolation* mechanism that prevents non-protected memory operations from accessing the safe region (§3.2.2). For performance reasons, we handle return addresses stored on the stack separately from the rest of the code pointers using a *safe stack* mechanism (§3.2.3).

CPI Static Analysis

We determine the set of sensitive pointers using type-based static analysis: a pointer is sensitive if its type is sensitive. Sensitive types are: pointers to functions, pointers to sensitive types, pointers to composite types (such as struct or array) that contains one or more members of sensitive types, or universal pointers (i.e., void*, char* and opaque pointers to forward-declared structs or classes). A programmer could additionally indicate, if desired, other types to be considered sensitive, such as struct ucred used in the FreeBSD kernel to store process UIDs and jail information. All code pointers that a compiler or runtime creates implicitly (such as return addresses, C++ virtual table pointers, and setjmp buffers) are sensitive as well.

Once the set of sensitive pointers is determined, we use static analysis to find all program instructions that manipulate these pointers. These instructions include pointer dereferences, pointer arithmetic, and memory (de-)allocation operations that calls to either (i) corresponding standard library functions, (ii) C++ new/delete operators, or (iii) manually annotated custom allocators.

The derived set of sensitive pointers is over-approximate: it may include universal pointers that never end up pointing to sensitive values at runtime. For instance, the C/C++ standard allows char* pointers to point to objects of any type, but such pointers are also used for C strings. As a heuristic, we assume that char* pointers that are passed to the standard libc string manipulation functions or that are assigned to point to string constants are not universal. Neither the over-approximation nor the char* heuristic affect the security guarantees provided by CPI: over-approximation merely introduces extra overhead, while heuristic errors may result in false violation reports (though we never observed any in practice).

Memory manipulation functions from libc, such as memset or memcpy, could introduce a lot of overhead in CPI: they take void* arguments, so a libc compiled with CPI would instrument all accesses inside the functions, regardless of whether they are operating on sensitive data or not. CPI's static analysis instead detects such cases by analyzing the real types of the arguments prior to being cast to void*, and the subsequent instrumentation pass handles them separately using type-specific versions of the corresponding memory manipulation functions.

We augmented type-based static analysis with a data-flow analysis that handles most practical cases of unsafe pointer casts and casts between pointers and integers. If a value v is ever cast to a sensitive pointer type within the function being analyzed, or is passed as an argument or returned to another function where it is cast to a sensitive pointer, the analysis considers v to be sensitive as well. This analysis may fail when the data flow between v and its cast to a sensitive pointer type cannot be fully recovered statically, which might cause false violation reports (we have not observed any during our evaluation). Such casts are a common problem for all pointer-based memory safety mechanisms for C/C++ that do not require source code modifications [101].

A key benefit of CPI is its selectivity: the number of pointer operations deemed to be sensitive is a small fraction of all pointer operations in a program. As we show in §6.1, for SPEC CPU2006, the CPI type-based analysis identifies for instrumentation 6.5% of all pointer accesses; this translates into a reduction of performance overhead of 16 – 44× relative to full memory safety.

Nevertheless, we still think CPI can benefit from more sophisticated analyses. CPI can leverage any kind of *points-to* static analysis, as long as it provides an over-approximate set of sensitive pointers. For instance, when extending CPI to also protect select non-code-pointer data, we think DSA [85, 86] could prove more effective.

CPI Instrumentation

CPI instruments a program in order to (i) ensure that all sensitive pointers are stored in a safe region, (ii) create and propagate metadata for such pointers at runtime, and (iii) check the metadata on dereferences of such pointers.

In terms of memory layout, CPI introduces a safe region in addition to the regular memory region (Figure 3.2). Storage space for sensitive pointers is allocated in both the safe region

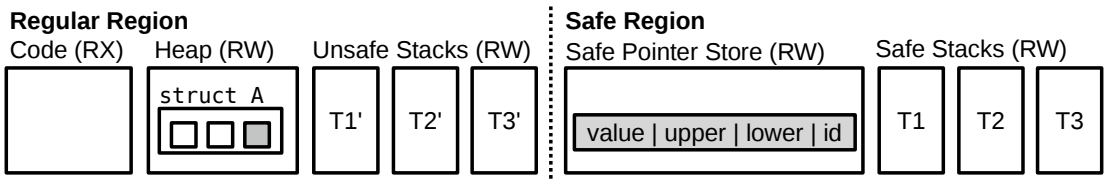


Figure 3.2 – CPI memory layout: The safe region contains the safe pointer store and the safe stacks. The location of a sensitive pointer on the left (shaded) remains unused, while the value of this pointer and its metadata are stored in the safe pointer store. The safe stacks T_1, T_2, T_3 have corresponding stacks T'_1, T'_2, T'_3 in regular memory to allocate unsafe stack objects.

(the *safe pointer store*) and the regular region (as usual); one of the two copies always remains unused. This is necessary for universal pointers (e.g., void*), which could be stored in either region depending on whether they are sensitive at run time or not, and also helps to avoid some compatibility issues that arise from the change in memory layout. The address in regular memory is used as an offset to to look up the value of a sensitive pointer in the safe pointer store.

The *safe pointer store* maps the address &p of sensitive pointer p, as allocated in the regular region, to the value of p and associated metadata. The metadata for p describes the target object on which p is based: lower and upper address bounds of the object, and a temporal id (see Figure 3.2). The layout of the safe pointer store is similar to metadata storage in SoftBounds+CETS [102], except that CPI *also* stores the value of p in the safe pointer store. Combined with the isolation of the safe region (§3.2.2), this allows CPI to guarantee full memory safety of all sensitive pointers without having to instrument all pointer operations.

The instrumentation step changes instructions that operate on sensitive pointers, as found by CPI's static analysis, to create and propagate the metadata directly following the based-on definition in §3.2.1. Instructions that explicitly take addresses of a statically allocated memory object or a function, allocate a new object on the heap, or take an address of a sub-object are instrumented to create metadata that describe the corresponding object. Instructions that compute pointer expressions are instrumented to propagate the metadata accordingly. Instructions that load or store sensitive pointers to memory are replaced with CPI intrinsic instructions (§3.2.2) that load or store both the pointer values and their metadata from/to the safe pointer store. In principle, call and return instructions also store and load code pointers, and so would need to be instrumented, but we instead protect return addresses using a safe stack (§3.2.3).

Every dereference of a sensitive pointer is instrumented to check at runtime whether it is safe, using the metadata associated with the pointer being dereferenced. Together with the restricted access to the safe region, this results in precise memory safety for all sensitive pointers.

Universal pointers (void* and char*) are stored in either the safe pointer store or the regular region, depending on whether they are sensitive at runtime or not. CPI instruments instructions that cast from non-sensitive to universal pointer types to assign special "invalid" metadata (e.g., with lower bound greater than the upper bound) for the resulting universal pointers. These pointers, as a result, would never be allowed to access the safe region. CPI intrinsics for universal pointers would only store a pointer in the safe pointer store if it had valid metadata, and only load it from the safe pointer store if it contained valid metadata for that pointer; otherwise, they would store/load from the regular region.

CPI can be configured to simultaneously store protected pointers in both the safe pointer store and regular regions, and check whether they match when loading them. In this debug mode, CPI detects all *attempts* to hijack control flow using non-protected pointer errors; in the default mode, such attempts are silently prevented. This debug mode also provides better compatibility with non-instrumented code that may read protected pointers (for example, callback addresses) but not write them.

Modern compilers contain powerful static analysis passes that can often prove statically that certain memory accesses are always safe. The CPI instrumentation pass precedes compiler optimizations, thus allowing them to potentially optimize away some of the inserted checks while preserving the security guarantees.

Isolating the Safe Region

The safe region can only be accessed via CPI intrinsic instructions, and they properly handle pointer metadata and the safe stack (§3.2.3). The mechanism for achieving this isolation is architecture-dependent.

On x86-32, we rely on hardware segment protection. We make the safe region accessible through a dedicated segment register, which is otherwise unused, and configure limits for all other segment registers to make the region inaccessible through them. The CPI intrinsics are then turned into code that uses the dedicated register and ensures that no other instructions in the program use that register. The segment registers are configured by the program loader, whose integrity we assume in our threat model; we also prevent the program from reconfiguring the segment registers via system calls. None of the programs we evaluated use the segment registers.

On other architectures, CPI can protect the safe region using precise software fault isolation (SFI) [24]. SFI requires that all memory operations in a program are instrumented, but the instrumentation is lightweight: it could be as small as a single `and` operation if the safe region occupies the entire upper half of the address space of a process. In our experiments, the additional overhead introduced by SFI was less than 5%.

On 64 bit architectures, CPI could also protect the safe region using randomization and information hiding. The fact that no addresses pointing into the safe region are ever stored in

the regular region is what makes perfect information hiding possible. For example, the x86-64 architecture no longer enforces the segment limits, however, it still provides two segment registers with configurable base addresses. Similarly to x86-32, we use one of these registers to point to the safe region, however, we choose the base address of the safe region at random and rely on preventing access to it through information hiding. Unlike classic ASLR though, our hiding is leak-proof: since the objects in the safe region are indexed by addresses allocated for them in the regular region, no addresses pointing into the safe region are ever stored in regular memory at any time during execution. When the size of the safe region is small (e.g., when the region is implemented as a hash table, as discussed in §3.4), the 48-bit address space of modern x86-64 CPUs makes guessing the safe region address impractical at least in some usage scenarios. Indeed, most failed guessing attempts would crash the program, and such frequent crashes can easily be detected by other means.

Upcoming Intel memory protection key extension will enable another hardware-enforced mechanism to protect the CPI safe region. We plan to evaluate the use of memory protection keys for this purpose once CPUs implementing them start appearing.

We further analyze the security and performance implications of the safe region protection mechanisms described above in §3.4.

Since sensitive pointers form a small fraction of all data stored in memory, the safe pointer store is highly sparse. To save memory, it can be organized as a hash table, a multi-level lookup table, or as a simple array relying on the sparse address space support of the underlying OS. We implemented and evaluated all three versions, and we discuss the fastest choice in §3.4.

In the future, we plan to leverage Intel MPX [70] for implementing the safe region, as described in §3.4.4.

3.2.3 The Safe Stack

CPI treats the stack specially, in order to reduce performance overhead and complexity. This is primarily because the stack hosts values that are accessed frequently, such as return addresses that are code pointers accessed on every function call, as well as spilled registers (temporary values that do not fit in registers and compilers store on the stack). Furthermore, tracking which of these values will end up at run time in memory (and thus need to be protected) vs. in registers is difficult, as the compiler decides which registers to spill only during late stages of code generation, long after CPI's instrumentation pass.

A key observation is that the safety of most accesses to stack objects can be checked statically during compilation, hence such accesses require no runtime checks or metadata. Most stack frames contain only memory objects that are accessed exclusively within the corresponding function and only through the stack pointer register with a constant offset. We therefore place all such proven-safe objects onto a *safe stack* located in the safe region. The safe stack can be accessed without any checks. For functions that have memory objects on their stack that

do require checks (e.g., arrays or objects whose address is passed to other functions), we allocate separate stack frames in the regular memory region. In our experience, less than 25% of functions need such additional stack frames (see Table 6.2). Furthermore, this fraction is much smaller among short functions, for which the overhead of setting up the extra stack frame is non-negligible.

The safe stack mechanism consists of a static analysis pass, an instrumentation pass, and runtime support. The analysis pass identifies, for every function, which objects in its stack frame are guaranteed to be accessed safely and can thus be placed on the safe stack; return addresses and spilled registers always satisfy this criterion. For the objects that do not satisfy this criterion, the instrumentation pass inserts code that allocates a stack frame for these objects on the regular stack. The runtime support allocates regular stacks for each thread and can be implemented either as part of the threading library, as we did on FreeBSD, or by intercepting thread create/destroy, as we did on Linux. CPI stores the regular stack pointer inside the thread control block, which is pointed to by one of the segment registers and can thus be accessed with a single memory read or write.

Our safe stack layout is similar to double stack approaches in ASR [12] and XFI [51], which maintain a separate stack for arrays and variables whose addresses are taken. However, we use the safe stack to enforce the CPI property instead of implementing software fault isolation. The safe stack is also comparable to language-based approaches like Cyclone [73] or CCured [103] that simply allocate these objects on the heap, but our approach has significantly lower performance overhead.

Compared to a shadow stack like in CFI [1], which duplicates return instruction pointers outside of the attacker's access, the CPI safe stack presents several advantages: (i) all return instruction pointers and most local variables are protected, whereas a shadow stack only protects return instruction pointers; (ii) the safe stack is compatible with uninstrumented code that uses just the regular stack, and it directly supports exceptions, tail calls, and signal handlers; (iii) the safe stack has near-zero performance overhead (§6.1.2), because only a handful of functions require extra stack frames, while a shadow stack allocates a shadow frame for every function call.

The safe stack can be employed independently from CPI, and we believe it can replace stack cookies [37] in modern compilers. By providing precise protection of all return addresses (which are the target of ROP attacks today), spilled registers, and some local variables, the safe stack provides substantially stronger security than stack cookies, while incurring equal or lower performance overhead and deployment complexity.

3.2.4 Code-Pointer Separation (CPS)

The code-pointer separation property trades some of CPI's security guarantees for reduced runtime overhead. This is particularly relevant to C++ programs with many virtual functions,

where the fraction of sensitive pointers instrumented by CPI can become high, since every pointer to an object that contains virtual functions is sensitive. We found that, on average, CPS reduces overhead by 4.3× (from 8.4% for CPI down to 1.9% for CPS), and in some cases by as much as an order of magnitude.

CPS further restricts the set of protected pointers to code pointers only, leaving pointers that point to code pointers uninstrumented. We additionally restrict the definition of based-on by requiring that a code pointer be based only on a control flow destination. This restriction prevents attackers from "forging" a code pointer from a value of another type, but still allows them to trick the program into reading or updating wrong code pointers.

CPS is enforced similarly to CPI, except (i) for the criteria used to identify sensitive pointers during static analysis, and (ii) that CPS does not need any metadata. Control-flow destinations (pointed to by code pointers) do not have bounds, because the pointer value must always match the destination exactly, hence no need for bounds metadata. Furthermore, they are typically static, hence do not need temporal metadata either (there are a few rare exceptions, like unloading a shared library, which are handled separately). This reduces the size of the safe region and the number of memory accesses when loading or storing code pointers. If the safe region is organized as a simple array, a CPS-instrumented program performs essentially the same number of memory accesses when loading or storing code pointers as a non-instrumented one; the only difference is that the pointers are being loaded or stored from the safe pointer store instead of their original location (universal pointer load or store instructions still introduce one extra memory access per such instruction). As a result, CPS can be enforced with low performance overhead.

CPS guarantees that (i) code pointers can only be stored to or modified in memory by code pointer store instructions, and (ii) code pointers can only be loaded by code pointer load instructions from memory locations to which previously a code pointer store instruction stored a value. Combined with the safe stack, CPS precisely protects return addresses. CPS is stronger than most CFI implementations [1, 137, 135], which allow any vulnerable instruction in a program to modify any code pointer; they only check that the value of a code pointer (when used in an indirect control transfer) points to a function defined in a program (for function pointers) or directly follows a call instruction (for return addresses). CPS guarantee (i) above restricts the attack surface, while guarantee (ii) restricts the attacker's flexibility by limiting the set of locations to which the control can be redirected—the set includes only entry points of functions whose addresses were explicitly taken by the program.

To illustrate this difference, consider the case of the Perl interpreter, which implements its opcode dispatch by representing internally a Perl program as a sequence of function pointers to opcode handlers and then calling in its main execution loop these function pointers one by one. CFI statically approximates the set of legitimate control-flow targets, which in this case would include all possible Perl opcodes. CPS, however, permits only calls through function pointers that are actually assigned. This means that a memory bug in a CFI-protected Perl

Atomic Types	a	::= `int` | $p*$
Pointer Types	p	::= a | s | f | `void`
Struct Types	s	::= `struct`$\{\ldots; a_i : id_i; \ldots\}$
LHS Expressions	lhs	::= x | $*lhs$ | $lhs.id$ | lhs–>id
RHS Expressions	rhs	::= i | $\&f$ | $rhs + rhs$ | lhs | $\&lhs$
		| $(a)\ rhs$ | `sizeof(p)` | `malloc(`rhs`)`
Commands	c	::= $c; c$ | $lhs = rhs$ | $f()$ | $(*lhs)()$

Figure 3.3 – The subset of C; x denotes local statically typed variables, id – structure fields, i – integers, and f – functions from a pre-defined set.

`sensitive`	`int`	::= *false*
`sensitive`	`void`	::= *true*
`sensitive`	f	::= *true*
`sensitive`	$p*$	::= `sensitive` p
`sensitive`	s	::= $\bigvee_{i \in fields\ of\ s}$ `sensitive` a_i

Figure 3.4 – The decision criterion for protecting types in CPI

interpreter may permit an attacker to divert control flow and execute any Perl opcode, whereas in a CPS-protected Perl interpreter the attacker could at most execute an opcode that exists in the running Perl program.

For C++ programs, CPS protects not only code pointers but also virtual table pointers. The abundance of virtual table pointers in most C++ programs gives attacker sufficient freedom to induce malicious program behavior by only chaining existing virtual functions through corresponding existing call sites [112]. Including virtual table pointers into the set of sensitive pointers protected by CPS prevents such attacks.

CPS provides strong control-flow integrity guarantees and incurs low overhead (§6.1). We found that it prevents all recent attacks designed to bypass CFI [61, 41, 22]. We consider CPS to be a solid alternative to CPI in those cases when CPI's (already low) overhead seems too high.

3.3 The Formal Model of CPI

This section presents a formal model and operational semantics of the CPI property and a sketch of its correctness proof. Due to the size and complexity of C/C++ specifications, we focus on a small subset of C that illustrates the most important features of CPI. Due to space limitations we focus on spatial memory safety. We build upon the formalization of spatial memory safety in SoftBound [101], reuse the same notation, and extend it to support applying

Operation	Semantics
$\texttt{read}_u\ M_u\ l$	return $M_u[l]$
$\texttt{write}_u\ M_u\ l\ v$	set $M_u[l] = v$
$\texttt{read}_s\ M_s\ l$	return $M_s[l]$, if l is allocated; return `none` otherwise
$\texttt{write}_s\ M_s\ l\ v_{(b,e)}$	set $M_s[l] = v_{(b,e)}$, if l is allocated; do nothing otherwise
$\texttt{write}_s\ M_s\ l\ \texttt{none}$	set $M_s[l] = \texttt{none}$, if l is allocated; do nothing otherwise
$\texttt{malloc}\ E\ i$	allocate a memory object of size i in both $E.M_u$ and $E.M_s$ (at the same address); fail when out of memory

Table 3.1 – Memory Operations in CPI

spatial memory safety to a subset of memory locations. The formalism can be easily extended to provide temporal memory safety, directly applying the CETS [102] mechanism to the safe memory region of the model. Figure 3.3 gives the syntax rules of the C subset we consider in this section. All valid programs must also pass type checking as specified by the C standard.

We define the runtime environment E of a program as a triple (S, M_u, M_s), where S maps variable identifiers to their respective atomic types and addresses, a regular memory M_u maps addresses to values (denoted as v and called regular values), and a safe memory M_s maps addresses to values with bounds information (denoted as $v_{(b,e)}$ and called safe values) or a special marker `none`. The bounds information specifies the lowest (b) and the highest (e) address of the corresponding memory object. M_u and M_s use the same addressing, but might contain distinct values for the same address. Some locations (e.g., of `void*` type) can store either safe or regular value and are resolved to either M_s or M_u at runtime.

The runtime provides the usual set of memory operations for M_u and M_s, as summarized in Table 3.1. M_u models standard memory, whereas M_s stores values with bounds and has a special marker for "absent" locations, similarly to the memory in SoftBound's [101] formalization. We assume the memory operations follow the standard behavior of read/write/malloc operations in all other respects, e.g., read returns the value previously written to the same location, malloc allocates a region of memory that is disjoint with any other allocated region, etc.

Enforcing the CPI property with low performance overhead requires placing most variables in M_u, while still ensuring that all pointers that require protection at runtime according to the CPI property are placed in M_s. In this formalization, we rely on type-based static analysis as defined by the `sensitive` criterion, shown on Figure 3.4. We say a type p is sensitive iff $\texttt{sensitive}\ p = \textit{true}$. Setting `sensitive` to true for all types would make the CPI operational semantics equivalent to the one provided by SoftBound and would ensure full spatial memory safety of all memory operations in a program.

The classification provided by the `sensitive` criterion is static and only determines which operations in a program to instrument. Expressions of sensitive types could evaluate to both safe or regular values at runtime, whereas expressions of regular types always evaluate to regular values. In particular, according to Figure 3.4, `void*` is sensitive and, hence, in

agreement with the C specification, values of that type can hold any pointer value at runtime, either safe or regular.

We extend the SoftBound definition of the result of an operation to differentiate between safe and regular values and left-hand-side locations:

$$\text{Results} \quad r ::= v_{(b,e)} \mid v \mid l_s \mid l_u \mid \texttt{OK} \mid \texttt{OutOfMem} \mid \texttt{Abort}$$

where $v_{(b,e)}$ and v are the safe (with bounds information) and, respectively, regular values that result from a right hand side expression, l_u and l_s are locations that result from a safe and regular left-hand-side expression, `OK` is a result of a successful command, and `OutOfMem` and `Abort` are error codes. We assume that all operational semantics rules of the language propagate these error codes up to the end of the program unchanged.

Using the above definitions, we now formalize the operational semantics of CPI through three classes of rules. The $(E, lhs) \Rightarrow_l l_s : a$ and $(E, lhs) \Rightarrow_l l_u : a$ rules specify how left hand side expressions are evaluated to a safe or regular locations, respectively. The $(E, rhs) \Rightarrow_r (v_{(b,e)}, E')$ and $(E, rhs) \Rightarrow_r (v, E')$ rules specify how right hand side expressions are evaluated to safe values with bounds or regular values, respectively, possibly modifying the environment through memory allocation (turning it from E to E'). Finally, the $(E, c) \Rightarrow_c (r, E')$ rules specify how commands are executed, possibly modifying the environment, where r can be either `OK` or an error code. We only present the rules that are most important for the CPI semantics, omitting rules that simply represent the standard semantics of the C language.

Bounds information is initially assigned when allocating a memory object or when taking a function's address (both operations always return safe values):

$$\frac{\texttt{address}(f) = l}{(E, \&f) \Rightarrow_r (l_{(l,l)})} \qquad \frac{(E, rhs) = i \quad \texttt{malloc}\ E\ i = (l, E')}{(E, \texttt{malloc}(i)) \Rightarrow_r (l_{(l,l+i)}, E')}$$

Taking the address of a variable from S if its type is sensitive is analogous. Structure field access operations either narrow bounds information accordingly, or strip it if the type of the accessed field is regular.

Type casting results in a safe value iff a safe value is cast to a sensitive type:

$$\frac{\texttt{sensitive}\ a' \quad (E, rhs) \Rightarrow_l v_{(b,e)} : a}{(E, (a')rhs) \Rightarrow_r (v_{(b,e)}, E)} \qquad \frac{\neg\texttt{sensitive}\ a' \quad (E, rhs) \Rightarrow_l v_{(b,e)} : a}{(E, (a')rhs) \Rightarrow_r (v, E)} \qquad \frac{(E, rhs) \Rightarrow_l v : a}{(E, (a')rhs) \Rightarrow_r (v, E)}$$

The next set of rules describes memory operations (pointer dereference and assignment) on

sensitive types and safe values:

$$\frac{\texttt{sensitive}\ a \quad (E, lhs) \Rightarrow_l l_s : a* \quad \texttt{read}_s(E.M_s)\, l_s = \texttt{some}\ l'_{(b,e)} \quad l' \in [b, e - \texttt{sizeof}(a)]}{(E, *lhs) \Rightarrow_l l'_s : a}$$

$$\frac{\texttt{sensitive}\ a \quad (E, lhs) \Rightarrow_l l_s : a* \quad \texttt{read}_s(E.M_s)\, l_s = \texttt{some}\ l'_{(b,e)} \quad l' \notin [b, e - \texttt{sizeof}(a)]}{(E, *lhs) \Rightarrow_l \texttt{Abort}}$$

$$\frac{\texttt{sensitive}\ a \quad (E, lhs) \Rightarrow_l l_s : a \quad (E, rhs) \Rightarrow_r v_{(b,e)} : a \quad E'.M_s = \texttt{write}_\texttt{s}(E.M_s)\, l_s\ v_{(b,e)}}{(E, lhs = rhs) \Rightarrow_c (\texttt{OK}, E')}$$

These rules are identical to the corresponding rules of SoftBound [101] and ensure full spatial memory safety of all memory objects in the safe memory. Only operations matching those rules are allowed to access safe memory M_s. In particular, any attempts to access values of sensitive types through regular lvalues cause aborts:

$$\frac{\texttt{sensitive}\ a \quad (E, lhs) \Rightarrow_l l_u : a*}{(E, *lhs) \Rightarrow_l \texttt{Abort}} \qquad \frac{\texttt{sensitive}\ a \quad (E, lhs) \Rightarrow_l l_u : a}{(E, lhs = rhs) \Rightarrow_c (\texttt{Abort}, E)}$$

Note that these rules can only be invoked if the value of the sensitive type was obtained by casting from a regular type using a corresponding type casting rule. Levee relaxes the casting rules to allow propagation of bounds information through certain right-hand-side expressions of regular types. This relaxation handles most common cases of unsafe type casting; it affects performance (inducing more instrumentation) but not correctness.

Some sensitive types (only `void*` in our simplified version of C), can hold regular values at runtime. For example, a variable of `void*` type can first be used to store a function pointer and subsequently re-used to store an `int*` value. The following rules handle such cases:

$$\frac{\texttt{sensitive}\ a \quad (E, lhs) \Rightarrow_l l_s : a* \quad \texttt{read}_s(E.M_s)\, l = \texttt{none} \quad \texttt{read}_u(E.M_u)\, l = l'}{(E, *lhs) \Rightarrow_l l'_u : a}$$

$$\frac{\texttt{sensitive}\ a \quad (E, lhs) \Rightarrow_l l_s : a \quad (E, rhs) \Rightarrow_r v : a \quad E'.M_u = \texttt{write}_\texttt{u}(E.M_u)\ l\ v \quad E'.M_s = \texttt{write}_\texttt{s}(E.M_s)\ l\ \texttt{none}}{(E, lhs = rhs) \Rightarrow_c (\texttt{OK}, E')}$$

Memory operations on regular types always access regular memory, without any additional runtime checks, following the unsafe memory semantics of C.

$$\frac{\neg\texttt{sensitive}\ a \quad (E, lhs) \Rightarrow_l l : a* \quad \texttt{read}_u(E.M_u)\, l = l'}{(E, *lhs) \Rightarrow_l l'_u : a}$$

$$\frac{\neg\texttt{sensitive}\ a \quad (E, lhs) \Rightarrow_l l : a \quad (E, rhs) \Rightarrow_r v : a \quad E'.M_u = \texttt{write}_\texttt{u}(E.M_u)\ l\ v}{(E, lhs = rhs) \Rightarrow_c (\texttt{OK}, E')}$$

These accesses to regular memory can go out of bounds but, given that $\texttt{read}_u$ and $\texttt{write}_u$ operations can only modify regular memory M_u, it does not violate memory safety of the safe

memory.

Finally, indirect calls abort if the function pointer being called is not safe:

$$\frac{(E, lhs) \Rightarrow_r l_s : f*}{(E, (*lhs)()) \Rightarrow_c (\texttt{OK}, E')} \qquad \frac{(E, lhs) \Rightarrow_r l_u : f*}{(E, (*lhs)()) \Rightarrow_c (\texttt{Abort}, E)}$$

Note that the operational rules for values that are safe at runtime are fully equivalent to the corresponding SoftBound rules [101]: the rules expressions are equal assuming `sensitive` a is `true` and, depending on the rule, either $\texttt{read}_\texttt{s}$ is not `none` or the right hand side value is sensitive. Therefore, under these conditions, these rules satisfy the SoftBound safety invariant which, as proven in [101], ensures memory safety for such values. According to the `sensitive` criterion and the safe location dereference and indirect function call rules above, all dereferences of pointers that require protection according to the CPI property are always safe at runtime, or the program aborts. Therefore, the operational semantics defined above indeed ensure the CPI property as defined in subsection 3.2.1.

3.4 Implementation

We implemented a CPI/CPS enforcement tool for C/C++, called Levee, on top of the LLVM 3.3 compiler infrastructure [90], with modifications to LLVM libraries, the clang compiler, and the compiler-rt runtime. To use Levee, one just needs to pass additional flags to the compiler to enable CPI (-fcpi), CPS (-fcps), or safe-stack protection (-fstack-protector-safe). Levee works on unmodified programs and supports Linux, FreeBSD, and Mac OS X in both 32-bit and 64-bit modes.

Levee can be downloaded from the project homepage http://levee.epfl.ch. The Safe Stack component of levee is already integrated upstream into the Clang compiler [111], and we plan to upstream the rest of our changes in the future as well. the upstream LLVM.

3.4.1 Analysis and instrumentation passes

CPI and CPS instrumentation passes: We implemented the static analysis and instrumentation for CPI as two LLVM passes, directly following the design from §3.2.2 and §3.2.2. The LLVM passes operate on the LLVM intermediate representation (IR), which is a low-level strongly-typed language-independent program representation tailored for static analyses and optimization purposes. The LLVM IR is generated from the C/C++ source code by clang, which preserves most of the type information that is required by our analysis, with a few corner cases. For example, in certain cases, clang does not preserve the original types of pointers that are cast to void* when passing them as an argument to memset or similar functions, which is required for the memset-related optimizations discussed in §3.2.2. The IR also does not distinguish between void* and char* (represents both as i8*), but this information is required

for our string pointers detection heuristic. We augmented clang to always preserve such type information as LLVM metadata.

Safe stack instrumentation pass: The safe stack instrumentation targets functions that contain on-stack memory objects that cannot be put on the safe stack. For such functions, it allocates a stack frame on the unsafe stack and relocates corresponding variables to that frame.

Given that most of the functions do not need an unsafe stack, Levee uses the usual stack pointer (rsp register on x86-64) as the safe stack pointer, and stores the unsafe stack pointer in the thread control block, which is accessible directly through one of the segment registers. When needed, the unsafe stack pointer is loaded into an IR local value, and Levee relies on the LLVM register allocator to pick the register for the unsafe stack pointer. Levee explicitly encodes unsafe stack operations as IR instructions that manipulate an unsafe stack pointer; it leaves all operations that use a safe stack intact, letting the LLVM code generator manage them. Levee performs these changes as a last step before code generation (directly replacing LLVM's stack-cookie protection pass), thus ensuring that it operates on the final stack layout.

Certain low-level functions modify the stack pointer directly. These functions include setjmp/longjmp and exception handling functions (which store/load the stack pointer), and thread create/destroy functions, which allocate/free stacks for threads. On FreeBSD we provide full-system CPI, so we directly modified these functions to support the dual stacks. On Linux, our instrumentation pass finds setjmp/longjmp and exception handling functions in the program and inserts required instrumentation at their call sites, while thread create/destroy functions are intercepted and handled by the Levee runtime.

3.4.2 Runtime support library

Most of the instrumentation by the above passes are added as intrinsic function calls, such as cpi_ptr_store() or cpi_memcpy(), which are implemented by Levee's runtime support library (a part of compiler-rt). This design cleanly separates the safe pointer store implementation from the instrumentation pass. In order to avoid the overhead associated with extra function calls, we ensure that some of the runtime support functions are always inlined. We compile these functions into LLVM bitcode and instruct clang to link this bitcode into every object file it compiles. Functions that are called rarely (e.g., cpi_abort(), called when a CPI violation is detected) are never inlined, in order to reduce the instruction cache footprint of the instrumentation.

We implemented and benchmarked several versions of the safe pointer store map in our runtime support library: a simple array, a two-level lookup table, and a hashtable. The array implementation relies on the sparse address space support of the underlying OS. Initially we found it to perform poorly on Linux, due to many page faults (especially at startup) and additional TLB pressure. Switching to superpages (2 MB on Linux) made this simple table the fastest implementation of the three. Note that, due the large virtual size of the simple table,

	Security		Overhead	
	CPI	CPS	CPI	CPS
Hardware segmentation	precise		8.4%	1.9%
Software fault isolation	precise		13.8%	7.0%
Information hiding				
- hashtable	16.6	20.7	9.7%	2.2%
- lookup table	15	17	8.9%	2.0%
- simple table	5	7	8.4%	1.9%

Table 3.2 – Security guarantees (either precise or number of entropy bits) and performance overhead (average on SPEC2006) of various implementations of CPI/CPS

the implementation based on it cannot be used in conjunction with randomization-based safe region isolation.

3.4.3 Safe region isolation

We implemented multiple mechanisms that efficiently enforce instruction-level isolation as required to protect the safe memory region: using hardware-enforced segmentation, software fault isolation, or randomization and information hiding. The security guarantees and performance implications of these mechanisms are summarized in Table 3.2, we discuss them in detail below. We focus on design choices behind each mechanism and ignore potential non-design bugs in the prototypes we released.

Hardware-Enforced Segmentation Based Implementation: On architectures that support hardware-enforced segmentation, CPI uses this feature directly to enforce instruction-level isolation. In such implementations, CPI dedicates a segment register to point to the safe memory region, and it enforces, at compile time, that only instructions instrumented with memory safety checks use this segment register. CPI configures all other segment registers, which are used by non-instrumented instructions, to prevent all accesses to the safe region through these segment registers on the hardware level.

Hardware-enforced segmentation is supported on x86-32 CPUs, but also on some x86-64 CPUs (see the Long Mode Segment Limit Enable flag), which demonstrates that adding segmentation to x86-64 CPUs is feasible, provided the techniques that could benefit from it prove to be indeed valuable.

This implementation of CPI is precise and imposes zero performance overhead on instructions that do not access sensitive pointers.

Software Fault Isolation Based Implementation: On architectures where hardware-enforced segmentation is not available (e.g., ARM and most of the x86-64 CPUs), the instruction-level isolation can be enforced using lightweight software fault isolation (SFI). In our implementation, we align the safe region in memory so that enforcing a pointer to not alias with it

can be done with a single bitmask operation (unlike more heavyweight SFI solutions, which typically add extra memory accesses and/or branches for each memory access in a program). Furthermore, accesses to the safe stack need not be instrumented, as they are guaranteed to be safe [82].

Our SFI-based implementation of CPI is precise, and SFI increases the overhead by less then 5% relative to hardware-enforced segmentation.

Information-Hiding Based Implementation: Another way to implement instruction-level isolation is based on randomization and information hiding. Such implementations exploit the guarantee of the CPI instrumentation that, in a CPI-instrumented program, no pointers into the safe region are ever stored outside of the safe region itself. When the base location of the safe region is randomized, the above guarantee implies that the attacker has to resort to random guessing in order to find the safe region, even in the presence of an arbitrary memory read vulnerability. On 64-bit architectures, most of the address space is unmapped and so, most of the failed guesses result in a crash. Such crashes, if frequent enough, can be detected by other means.

The actual expected number of crashes required to find the location of the safe region by random guessing is determined by the size of the safe region and the size of the address space. Today's mainstream x86-64 CPUs provide 2^{48} bytes address space (while the architecture itself envisions future extensions up to 2^{64} bytes). Half of the address space is usually occupied by the OS kernel, which leaves 2^{47} bytes for applications.

As explained above, the safe region stores a map that, for each sensitive pointer, maps the location that the pointer would occupy in the memory of a non-instrumented program to a tuple of the pointer value and its metadata. On 64-bit CPUs, each entry of this map occupies 32 bytes and, due to pointer alignment requirements, represents 8 bytes of program memory. The expected number of entries depends on the program memory usage, the fraction of sensitive pointers, and the data structure that is used to store this map.

We released three versions of our information-hiding based CPI implementation that use either a hashtable, a two-level lookup table, or a simple table to organize the safe region [82]. Although all these safe region organizations are compatible with hardware-enforced segmentation and software based fault isolation implementations, the choice of the organization has the highest impact on the information-hiding based implementation. We analyze this impact below. We estimate the size of the safe region and the expected number of crashes required to find its location for each of the versions below. For the purpose of this estimation, we assume a program uses 1GB of memory, 8% of which stores sensitive pointers (which is consistent with the experimental evaluation in [82]), which amounts to $1\,\text{GByte} \times 8\% / 8\,\text{bytes} \approx 2^{23.4}$ sensitive pointers in total.

Hashtable This implementation is based on a linearly-probed lookup table with a bitmask-and-shift based hash function, which, due to sparsity of sensitive pointers in program memory,

performs well with load factors of up to 0.5. Conservatively assuming a load factor of 0.25, the hashtable would occupy $2^{23.4}/0.25 \times 32 = 2^{30.4}$ bytes of memory. Randomizing the hashtable location can provide up to $47 - 30.4 = 16.6$ bits of entropy, requiring $2^{15.6} \approx 51{,}000$ crashes on average to guess it. In most systems, that many crashes can be detected externally, making the attack infeasible.

Two-level lookup table This implementation organizes the safe region similar to page tables, using the higher 23 bits of the address as an index in the directory, and the lower 22 bits as an index in a subtable (the lowest 3 bits are zero due to alignment). Each subtable takes 32×2^{22} bytes and describes a 8×2^{22} bytes region of the address space. Assuming sensitive pointers are uniformly distributed across the 1GB of continuous program memory, CPI will allocate $1\,\text{GByte}/(8 \times 2^{22}) \times 32 \times 2^{22} = 2^{32}$ bytes for the subtables. Randomizing the subtables locations gives $47 - 32 = 15$ bits of entropy, requiring 2^{14} crashes on average to guess. Note that the attacker will find a random one among multiple subtables, and finding usable code pointers in it requires further guessing. This attack is thus also infeasible in many practical cases.

Simple table This simple implementation allocates a fixed-size region of 2^{42} bytes for the safe region that maps addresses linearly. This implementation would give only $47 - 42 = 5$ bits of entropy. The location of the simple table in this implementation can be guessed while causing only 16 crashes on average. This level of protection is not sufficient for most practical cases.

Code-Pointer Separation, unlike full CPI, does not require any metadata and has fewer sensitive pointers (by 8.5× on average [82]). This increases the number of expected crashes by 17× for the hashtable-based implementation, and by 4× for the other two implementations.

The information-hiding-based implementation of CPI that uses a hashtable to organize the safe region provides probabilistic security guarantees with $2^{16.6}$ bits of entropy (or $2^{20.7}$ for CPS). We believe that, in certain practical use cases, this number of crashes, especially given the uniform pattern of these crashes, can be detected automatically by external means.

In our evaluations, all three versions of information-hiding-based implementation have performance overhead comparable to the hardware-enforced segmentation.

3.4.4 Discussion

Binary level functionality: Some code pointers in binaries are generated by the compiler and/or linker, and cannot be protected on the IR level. Such pointers include the ones in jump tables, exception handler tables, and the global offset table. Bounds checks for the jump tables and the exception handler tables are already generated by LLVM anyway, and the tables themselves are placed in read-only memory, hence cannot be overwritten. We rely on the standard loader's support for read-only global offset tables, using the existing `RTLD_NOW` flag.

Limitations: The CPI design described in §3.2 includes both spatial and temporal memory

safety enforcement for sensitive pointers, however, our current prototype implements spatial memory safety only. It can be easily extended to enforce temporal safety by directly applying the technique described in [102] for sensitive pointers.

Levee currently supports Linux, FreeBSD and Mac OS user-space applications. We believe Levee can be ported to protect OS kernels as well. Related technical challenges include integration with the kernel memory management subsystem and handling of inline assembly.

CPI and CPS require instrumenting all code that manipulates sensitive pointers; non-instrumented code can cause unnecessary aborts. Non-instrumented code could come from external libraries compiled without Levee, inline assembly, or dynamically generated code. Levee can be configured to simultaneously store sensitive pointers in both the safe and the regular regions, in which case non-instrumented code works fine as long as it only reads sensitive pointers but doesn't write them.

Inline assembly and dynamically generated code can still update sensitive pointers if instrumented with appropriate calls to the Levee runtime, either manually by a programmer or directly by the code generator.

Dynamically generated code (e.g., for JIT compilation) poses an additional problem: running the generated code requires making writable pages executable, which violates our threat model (this is a common problem for most control-flow integrity mechanisms). One solution is to use hardware or software isolation mechanisms to isolate the code generator from the code it generates.

Sensitive data protection: Even though the main focus of CPI is control-flow hijack protection, the same technique can be applied to protect other types of sensitive data. Levee can treat programmer-annotated data types as sensitive and protect them just like code pointers. CPI could also selectively protect individual program variables (as opposed to types), however, it would require replacing the type-based static analysis described in §3.2.2 with data-based points-to analysis such as DSA [85, 86].

Future MPX-based implementation: Intel announced a hardware extension, Intel MPX, to be used for hardware-enforced memory safety [69]. It is proposed as a testing tool, probably due to the associated overhead; no overhead numbers are available at the time of writing.

We believe MPX (or similar) hardware can be re-purposed to enforce CPI with lower performance overhead than our existing software-only implementation. MPX provides special registers to store bounds along with instructions to check them, and a hardware-based implementation of a pointer metadata store (analogous to the safe pointer store in our design), organized as a two-level lookup table. Our implementation can be adapted to use these facilities once MPX-enabled hardware becomes available. We believe that a hardware-based CPI implementation can reduce the overhead of a software-only CPI in much the same way as HardBound [43] or Watchdog [100] reduced the overhead of SoftBound.

Adopting MPX for CPI might require implementing metadata loading logic in software. Like CPI, MPX also stores the pointer value together with the metadata. However, being a testing tool, MPX chooses compatibility with non-instrumented code over security guarantees: it uses the stored pointer value to check whether the original pointer was modified by non-instrumented code and, if yes, resets the bounds to $[0,\infty]$. In contrast, CPI's guarantees depend on preventing any non-instrumented code from ever modifying sensitive pointer values.

4 Automated Device Driver Testing with Selective Symbolic Execution

In this chapter we present DDT, a device driver testing system that empowers end-users to test closed-source device drivers.

DDT takes as input a binary device driver and outputs a report of found bugs, along with execution traces for each bug. The input driver is loaded in its native, unmodified environment, which consists of the OS kernel and the rest of the software stack above it. DDT then exercises automatically the driver along as many code paths as possible, and checks for undesired properties. When an error or misbehavior is detected, DDT logs the details of the path exploration along with an executable trace. This can be used for debugging, or merely as evidence to prove the presence of the bug.

DDT has two main components: a set of pluggable bug checkers and a driver exerciser (Figure 4.1). The exerciser is in charge of steering the driver down various execution paths—just like a personal trainer, it forces the driver to exercise all the various ways in which it can run. The dynamic checkers watch the execution and flag undesired driver behaviors along the executed paths. When they notice a bug, they ask the exerciser to produce information on how to reach that same situation again.

DDT provides a default set of checkers, and this set can be extended with an arbitrary number of other checkers for both safety and liveness properties (see §4.1.1). Currently, DDT detects the following types of bugs: memory access errors, including buffer overflows; race conditions and deadlocks; incorrectly handled interrupts; accesses to pageable memory when page faults are not allowed; memory leaks and other resource leaks; mishandled I/O requests (e.g., setting various I/O completion flags incorrectly); any action leading to kernel panic; and incorrect uses of kernel APIs.

These default checkers catch the majority of defects in the field. Ganapathi et al. found that the top driver problems causing crashes in Windows were 45% memory-related (e.g., bad pointers), 15% poorly handled exceptions, 13% infinite loops, and 3% unexpected traps [55]. A Microsoft report [99] found that, often, drivers crash the system due to not checking for error

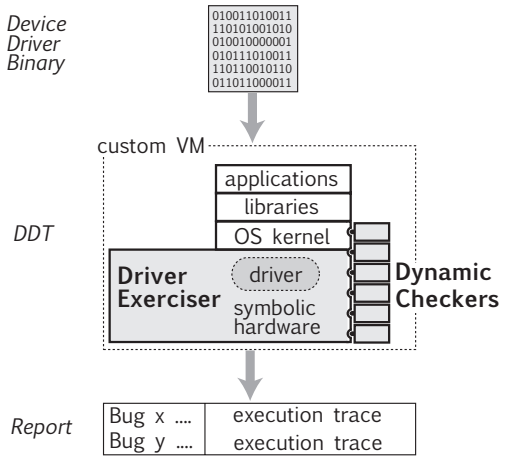


Figure 4.1 – DDT's VM-based architecture.

conditions following a call to the kernel. It is hypothesized that this is due to programmers copy-pasting code from the device driver development kit's succinct examples.

Black-box testing of closed-source binary device drivers is difficult and typically has low code coverage. This has two main reasons: First, it is hard to exercise the driver through the many layers of the software stack that lie between the driver's interface and the application interface. Second, closed-source programs are notoriously hard to test as a black box. The classic approach to testing such drivers is to try to produce inputs that exercise as many paths as possible and (perhaps) check for high-level properties (e.g., absence of kernel crashes) during those executions. Considering the wide range of possible inputs and system events that are hard to control (e.g., interrupts), this approach exercises relatively few paths, thus offering few opportunities to find bugs.

DDT uses selective symbolic execution [30] of the driver binary to automatically take the driver down as many paths as possible; the checkers verify desired properties along these paths. Symbolic execution [78, 19, 20] consists of providing a program with symbolic inputs (e.g., α or β) instead of concrete ones (e.g., 6 or "abc"), and letting these values propagate as the program executes, while tracking path constraints (e.g., $\beta = \alpha + 5$). When a symbolic value is used to decide the direction of a conditional branch, symbolic execution explores all feasible alternatives. On each branch, a suitable path constraint is added on the symbolic value to ensure its set of possible values satisfies the branch condition (e.g., $\beta < 0$ and $\beta >= 0$, respectively). Selective symbolic execution enables the symbolic execution of one piece of the software stack (the device driver, in our case) while the rest of the software runs concretely.

A key challenge is keeping the symbolic and the concrete portions of the execution synchronized. DDT supplies the driver with symbolic values on the calls from the kernel to the driver (§4.1.2) as well as on the returns from the hardware to the driver (§4.1.3), thus enabling an underlying symbolic execution engine to steer the driver on the various possible paths. When the driver returns values to a kernel-originated call, or when the driver calls into the ker-

nel, parameters and driver are converted so that execution remains consistent, despite the alternation of symbolic and concrete execution.

DDT's *fully symbolic hardware* enables testing drivers even when the corresponding hardware device is not available. DDT never calls the actual hardware, but instead replaces all hardware reads with symbolic values, and discards all writes to hardware. Being able to test a driver without access to the hardware is useful, for example, for certification companies that cannot buy all the hardware variants for the drivers they test, or for consumers who would rather defer purchasing the device until they are convinced the driver is trustworthy.

Symbolic hardware also enables DDT to explore paths that are hard to test without simulators or specialized hardware. For example, many devices rely on interrupts to signal completion of operations to the device driver. DDT uses *symbolic interrupts* to inject such events at the various crucial points during the execution of the driver. Symbolic interrupts allow DDT to test different code interleavings and detect bugs like the race conditions described in §6.2.1.

DDT provides evidence of the bug and the means to debug it: a complete trace of the execution plus concrete inputs and system events that make the driver re-execute the buggy path in a regular, non-DDT environment.

The rest of the chapter is structured as follows: we describe the DDT design in detail (§4.1), present our current DDT prototype for Windows drivers (§4.2), and discusses the limitations of our current prototyp (§4.3).

4.1 DDT Design

We now present DDT's design, starting with the types of bugs DDT looks for (§4.1.1), an overview of how drivers are exercised (§4.1.2), a description of fully symbolic hardware (§4.1.3), the use of annotations to extend DDT's capabilities (§4.1.4), and finally we show how generated traces are used to replay bugs and fix them (§4.1.5).

4.1.1 Detecting Undesired Behaviors

DDT uses two methods to detect failures along exercised paths: dynamic verification done by DDT's virtual machine (§4.1.1) and failure detection inside the guest OS (§4.1.1). VM-level checks are targeted at properties that require either instrumentation of driver code instructions or reasoning about multiple paths at a time. Guest OS-level checks leverage existing stress-testing and verification tools to catch bugs that require deeper knowledge of the kernel APIs. Most guest OS-level checks can be performed at the VM level as well, but it is often more convenient to write and deploy OS-level checkers.

Virtual Machine-Level Checks

Memory access verification in DDT is done at the VM level. On each memory access, DDT checks whether the driver has sufficient permissions to access that memory. For the purpose of access verification, DDT treats the following memory regions as accessible to drivers:

- Dynamically allocated memory and buffers;
- Buffers passed to the driver, such as network packets or strings from the Windows registry;
- Global kernel variables that are implicitly accessible to drivers;
- Current driver stack (accesses to memory locations below the stack pointer are prohibited, because these locations could be overwritten by an interrupt handler that saves context on the stack);
- Executable image area, i.e., loadable sections of the driver binary with corresponding permissions;
- Hardware-related memory areas (memory-mapped registers, DMA memory, or I/O ranges).

In order to track these memory regions, DDT hooks the kernel API functions and driver entry points. Every time the hooked functions are called, DDT analyzes their arguments to determine which memory was granted to (or revoked from) the driver. The required knowledge about specific kernel APIs can be provided through lightweight API annotations (see §4.1.4).

Beyond memory safety, DDT's simultaneous access to multiple execution paths (by virtue of employing symbolic execution) enables the implementation of bug detection techniques that reason about the code globally in terms of paths, such as infinite loop detection [136].

Guest Operating System-Level Checks

In addition to VM-level checkers, DDT can also reuse off-the-shelf runtime verification tools. These tools perform in-guest checking, oblivious to exactly how the driver is being driven along the observed execution paths. Since these tools are usually written by OS developers (e.g., for driver certification programs, like Microsoft's WHQL [98]), they can detect errors that require deep knowledge of the OS and its driver API.

When they find a bug, these dynamic tools typically crash the system to produce an error report containing a memory dump. DDT intercepts such premeditated crashes and reports the bug information to the user. DDT helps the runtime checkers find more bugs than they would do under normal concrete execution, because it symbolically executes the driver along many more paths.

DDT's modular architecture (Figure 4.1) allows reusing such tools without adaptation or porting. This means that driver developers' custom test suites can also be readily employed.

Moreover, given DDT's design, such tools can be inserted at any level in the software stack, either in the form of device drivers or as software applications.

DDT can also automatically leverage kernel assertion checks, when they are present. For example, the checked build version of Windows contains many consistency checks—with DDT, these assertions get a better chance of being exercised along different paths.

4.1.2 Exercising the Driver: Kernel/Driver Interface

DDT implements selective symbolic execution [30], a technique for seamless transfer of system state between symbolic and concrete phases of execution. DDT obtains similar properties to running the entire system symbolically, while in fact only running the driver symbolically. The transfer of state between phases is governed by a set of conversion hints (see §4.1.4). Using selective symbolic execution enables DDT to execute the driver within its actual environment, as opposed to requiring potentially incomplete models thereof [10, 19].

A typical driver is composed of several entry points. When the OS loads the driver, it calls its main entry point, similarly to a shell invoking the `main()` function of a program. This entry point registers with the kernel the driver's other entry points. For example, a typical driver would register `open`, `read`, `write`, and `close` entry points, which are then called by the OS when a user-mode application makes use of the driver.

When the kernel calls a driver's entry point, DDT transfers system state to a symbolic execution engine. It converts entry point arguments, and possibly other parts of concrete system state, to symbolic values, according to the annotations described in §4.1.4. For example, when the kernel calls the `SendPacket` function in a NIC driver, DDT makes the content of the network packet symbolic, to explore all the paths that depend on the packet's type.

When a driver calls a kernel function, DDT selects feasible values (at random) for its symbolic arguments. For example, if the driver calls the `AllocatePool` function with a symbolic length argument, DDT selects some concrete value *len* for the length that satisfies current constraints. However, this concretization subjects all subsequent paths to the constraint that length must equal *len*, and this may disable otherwise-feasible paths. Thus, DDT keeps track of all such concretization-related constraints and, if at some point in the future this constraint limits a choice of paths, DDT backtracks to the point of concretization, forks the entire machine state, and repeats the kernel call with different feasible concrete values, which could re-enable the presently unexplorable path.

To minimize overhead, DDT does concretization on-demand, i.e., delays it as long as possible by tracking symbolic values when executing in concrete mode and concretizing them only when they are actually read. This way, symbolic values that are not accessed by concretely running code are never concretized. In particular, all private driver state and buffers that are treated as opaque by the kernel end up being preserved in their symbolic form.

4.1.3 Exercising the Driver: Symbolic Hardware

DDT requires neither real hardware nor hardware models to test drivers—instead, DDT uses symbolic hardware. A symbolic device in DDT ignores all writes to its registers and produces symbolic values in response to reads. These symbolic values cause drivers to explore paths that depend on the device output.

Symbolic hardware produces symbolic interrupts, i.e., interrupts with a symbolic arrival time. Reasoning about interrupt arrival symbolically offers similar benefits to reasoning about program inputs symbolically: the majority of interrupt arrival times are equivalent to each other, so only one arrival time in each equivalence class need be produced. If a block of code does not read/write system state that is also read/written by the interrupt handler, then executing the interrupt handler at any point during the execution of that block has the same end result.

Currently, DDT implements a simplified model of symbolic interrupts. It symbolically delivers interrupts on each crossing of the kernel/driver boundary (i.e., before and after each kernel API call, and before and after each driver entry point execution). While not complete, we found that this strategy produces good results, because many important changes in driver state are related to crossing the kernel/driver interface.

Symbolic hardware with symbolic interrupts may force the driver on paths that are not possible in reality with correct hardware. For example, a symbolic interrupt may be issued after the driver instructed the device not to issue interrupts (e.g., by writing a control register). A correctly functioning device will therefore not deliver that interrupt. The natural solution would be to include the enabled/disabled interrupts status in the path constraints, and prevent interrupts from occurring when this is not possible. However, recent work [74] has shown that hardware often malfunctions, and that drivers must be sufficiently robust to handle such behavior anyway.

More generally, DDT's ability to test drivers against hardware failures is important, because chipsets often get revised without the drivers being suitably updated. Consider a device that returns a value used by the driver as an array index. If the driver does not check the bounds (a common bug [74]) and a revised version of the chipset later returns a greater value, then the obsolete driver could experience an out-of-bounds error.

4.1.4 Enabling Rich Driver/Environment Interactions

Device drivers run at the bottom of the software stack, sandwiched between the kernel and hardware devices. The layers surrounding a driver are complex, and the different classes of device drivers use many different kernel subsystems. For instance, network, audio, and graphics drivers each use different kernel services and interfaces.

One may be tempted to run drivers in isolation for purposes of testing. Unfortunately, this requires an abstraction layer between the drivers and the rest of the stack, and building this layer is non-trivial. For example, testing a network driver would require the testbed to provide well-formed data structures when returning from a packet allocation function called by the driver.

DDT tests drivers by symbolically executing them in conjunction with the real kernel binary. By using the actual software stack (and thus the real kernel) instead of a simplified abstract model of it, DDT ensures that the device drivers get tested with the exact kernel behavior they would experience in reality. To this end, DDT needs to mediate the interactions with the layers around the driver in a way that keeps the symbolic execution of the driver consistent with the concrete execution of the kernel.

DDT performs various conversions between the symbolic and concrete domains. In its default mode, in which no annotations are used, DDT converts symbolic arguments passed to kernel functions into legal random concrete values and uses symbolic hardware, including symbolic interrupts. Driver entry point arguments are not touched. These conversions, however, can be fine-tuned by annotating API functions and driver entry points.

Extending DDT with Interface Annotations

DDT provides ways for developers to encode their knowledge of the driver/kernel API in annotations that improve DDT's achievable code coverage and bug finding abilities. Annotations allow DDT to detect not only low-level errors, but also logical bugs. Annotations are a one-time effort on the part of OS developers, testers, or a broader developer community.

The idea of encoding API usage rules in annotations is often used by model checking tools, with a recent notable example being SLAM [10]. However, DDT's annotations are lighter weight and substantially easier to write and keep up-to-date than the API models used by previous tools: preparing DDT annotations for the whole NDIS API, which consists of 277 exported functions, took about two weeks of on-and-off effort; preparing annotations for those 54 functions in the WDM API that were used by our sound drivers took one day.

DDT annotations are written in C and compiled to LLVM bitcode [84], which is then loaded by DDT at runtime and run in the context of QEMU-translated code, when necessary. The annotation code has direct access to, and control over, the guest system's state. Additionally, it can use a special API provided by DDT to create symbolic values and/or manipulate execution state.

The following annotation introduces positive integer symbolic values when the driver reads a configuration parameter from the Windows registry:

```
 1 void NdisReadConfiguration_return(CPU* cpu) {
 2   if(*((PNDIS_STATUS) ARG(cpu, 0)) == 0
 3            && ARG(cpu, 4) == 1) {
 4     int symb = ddt_new_symb_int();
 5     if(symb >= 0)
 6       ((PNDIS_CONFIGURATION_PARAMETER)
 7         ARG(cpu, 1))->IntegerData = symb;
 8     else ddt_discard_state();
 9   }
10 }
```

This sample annotation function is invoked on the return path from `NdisReadConfiguration` (hence its name—line 1). It checks whether the call returned successfully (line 2) and whether the type of the value is integer (line 3). It then creates an unconstrained symbolic integer value using DDT's special API (line 4), after which it checks the value (line 5) and discards the path on which *symb* is not a positive integer (line 8).

DDT annotations fall into four categories:

Concrete-to-symbolic conversion hints apply to driver entry points' arguments and to return values from kernel functions called by the driver. They encode contracts about what constitute reasonable arguments or return values. For example, a memory allocation function can either return a valid pointer or a null pointer, so the annotation would instruct DDT to try both the originally returned concrete pointer, as well as the null-pointer alternative. The absence of this kind of conversion hints will cause DDT not to try all reasonable classes of values, which results solely in decreased coverage, i.e., false negatives.

Symbolic-to-concrete conversion hints specify the allowed set of values for arguments to kernel API functions called by drivers. They include various API usage rules that, if violated, may lead to crashes or data corruption. When a call to such an annotated function occurs, DDT verifies that all incorrect argument values are ruled out by the constraints on the current path; if not, it flags a potential bug. The absence of such annotations can lead DDT to concretize arguments into some values that are consistent with the path constraints (thus feasible in a real execution) but not uncover potential bugs (if the values happen to be OK according to the unspecified API usage rules). In other words, they can lead to false negatives, but not to false positives.

Resource allocation hints specify whether invoking an entry point or calling a kernel function grants or revokes the driver's access to any memory or other resources. This information is used to verify that the driver accesses only resources that the kernel explicitly allows it to access. It is also used to verify that all allocated resources are freed on exit paths. The absence of memory allocation hints can lead to false positives, but can be avoided, if necessary, by switching to a coarse-grained memory access verification scheme (as used, for instance, in Microsoft's Driver Verifier [98]).

Kernel crash handler hook: This annotation informs DDT of the address of the guest kernel's

crash handler, as well as how to extract the crash information from memory. This annotation enables DDT to intercept all crashes when running the kernel concretely, such as the "blue screen of death" (BSOD) on Windows. This annotation is relied upon in our DDT prototype to cooperate with the Microsoft Driver Verifier's dynamic checkers.

Alternative Approaches

We have gone to great lengths to run the drivers in a real environment and avoid abstract modeling. Is it worth it?

One classic approach to ensuring device driver quality is stress-testing, which is how Microsoft certifies its third-party drivers [98]. However, this does not catch all bugs. As we shall see in the evaluation, even Microsoft-certified drivers shipped with Windows have bugs that cause the kernel to crash. However, powerful static analysis tools [10] can reason about corner-case conditions by abstracting the driver under test, without actually running it. Since static analysis does not run any code per se, it requires modeling the driver's environment.

We believe environment modeling generally does not scale, because kernels are large and evolve constantly. Modeling the kernel/driver API requires manual effort and is error prone. According to [10], developing around 60 API usage rules for testing Windows device drivers took more than three years. It also required many iterations of refinement based on false positives found during evaluation. In the end, the resulting models are only an approximation of the original kernel code, thus leading to both false negatives and, more importantly, false positives. A test tool that produces frequent false positives discourages developers from using it.

In contrast, we find DDT's annotations to be straightforward and easy to maintain. Moreover, if they are perceived by developers as too high of a burden, then DDT can be used in its default mode, without annotations.

Testing device drivers often requires access to either the physical device or a detailed model of it. For drivers that support several physical devices, testing must be repeated for each such device. In contrast, symbolic hardware enables not only testing drivers without a physical device, but also testing them against hardware bugs or corner cases that are hard to produce with a real device.

4.1.5 Verifying and Replaying Bugs

When DDT finishes testing a driver, it produces a detailed report containing all the bugs it found. This report consists of all faulty execution paths and contains enough information to accurately replay the execution, allowing the bug to be reproduced on the developer's or consumer's machine.

DDT's bug report is a collection of traces of the execution paths leading to the bugs. These traces contain the list of program counters of the executed instructions up to the bug occurrence, all memory accesses done by each instruction (address and value) and the type of the access (read or write). Traces contain information about creation and propagation of all symbolic values and constraints on branches taken. Each branch instruction has a flag indicating whether it forked execution or not, thus enabling DDT to subsequently reconstruct an execution tree of the explored paths; each node in the tree corresponds to a machine state. Finally, DDT associates with each failed path a set of concrete inputs and system events (e.g., interrupts) that take the driver along that path. The inputs are derived from the symbolic state by solving the corresponding path constraints [57, 20].

A DDT trace has enough information to replay the bug in the DDT VM. Each trace starts from an initial state (a "hibernated" snapshot of the system) and contains the exact sequence of instructions and memory accesses leading to the crash or hang. The traces are self-contained and directly executable. The size of these traces rarely exceeds 1 MB per bug, and usually they are much smaller. We believe DDT traces can easily be adapted to work with existing VM replay tools [47, 119, 79].

DDT also post-processes these traces off-line, to produce a palatable error report. DDT reconstructs the tree of execution paths and, for each leaf state that triggered a bug, it unwinds the execution path by traversing the execution tree to the root. Then it presents the corresponding execution path to the developer. When driver source code is available, DDT-produced execution paths can be automatically mapped to source code lines and variables, to help developers better visualize the buggy behavior.

For bugs leading to crashes, it is also possible to extract a Windows crash dump that can be analyzed with WinDbg [98]—since each execution state maintained by DDT is a complete snapshot of the system, this includes the disk where the OS saved the crash dump. It is also worth noting that DDT execution traces can help debuggers go backwards through the buggy execution.

In theory, DDT traces could be directly executed outside the VM (e.g., in a debugger) using a natively executing OS, since the traces constitute slices through the driver code. The problem, though, is that the physical hardware would need to be coerced into providing the exact same sequence of interrupts as in the trace—perhaps this could be done with a PCI-based FPGA board that plays back a trace of interrupts. Another challenge is providing the same input and return values to kernel calls made by the driver—here DDT could leverage existing hooking techniques [17, 66] to intercept and modify these calls during replay. Finally, replaying on a real machine would involve triggering asynchronous events at points equivalent to those saved in the traces [131].

4.1.6 Analyzing Bugs

Execution traces produced by DDT can also help understand the cause of a bug. For example, if an assertion of a symbolic condition failed, execution traces can identify on what symbolic values the condition depended, when during the execution were they created, why they were created, and what concrete assignment of symbolic values would cause the assertion to fail. An assertion, bad pointer access, or a call that crashes the kernel might depend indirectly on symbolic values, due to control flow-based dependencies; most such cases are also identifiable in the execution traces.

Based on device specifications provided by hardware vendors, one can decide whether a bug can only occur when a device malfunctions. Say a DDT symbolic device returned a value that eventually led to a bug; if the set of possible concrete values implied by the constraints on that symbolic read does not intersect the set of possible values indicated by the specification, then one can safely conclude that the observed behavior would not have occurred unless the hardware malfunctioned.

One could write tools to automate the analysis and classification of bugs found by DDT, even though doing this manually is not hard. They could provide both user-readable messages, like "driver crashes in low-memory situations," and detailed technical information, like "`AllocateMemory` failed at location pc_1 caused a null pointer dereference at some other location pc_2.

4.2 DDT Implementation

We now describe our implementation of a DDT prototype for Windows device drivers (§4.2.1), which can be used by both developers and consumers to test binary drivers before installing them. We also show how to trick Windows into accepting DDT's symbolic hardware (§4.2.2) and how to identify and exercise the drivers' entry points (§4.2.3). Although Windows-specific, these techniques can be ported to other platforms as well.

4.2.1 DDT for Microsoft Windows

DDT uses a modified QEMU [11] machine emulator together with a modified version of the Klee symbolic execution engine [19]. DDT can run a complete, unmodified, binary software stack, comprising Windows, the drivers to be tested, and all associated applications.

Doing VM-Based Symbolic Execution

QEMU is an open-source machine emulator that supports many different processor architectures, like x86, SPARC, ARM, PowerPC, and MIPS. It emulates the CPU, memory, and devices using dynamic binary translation. QEMU's support of multiple architectures makes DDT

available to more than just x86-based platforms.

DDT embeds an adapted version of Klee. To symbolically execute a program, one first compiles it to LLVM bitcode [84], which Klee can then interpret. Klee employs various constraint solving optimizations and coverage heuristics, which make it a good match for DDT.

To use Klee, we extended QEMU's back-end to generate LLVM bitcode. QEMU translates basic blocks from the guest CPU instruction set to a QEMU-specific intermediate representation—we translate from this intermediate representation to LLVM on the fly. The generated LLVM bitcode can be directly interpreted by Klee.

QEMU and Klee have different representations of program state, which have to be kept separate yet synchronized. In QEMU, the state is composed of the virtual CPU, VM physical memory, and various virtual devices. We encapsulate this data in Klee memory objects, and modified QEMU to use Klee's routines to manipulate the VM's physical memory. Thus, whenever the state of the CPU is changed (e.g., register written) or a device is accessed (e.g., interrupt controller registers are set), both QEMU and Klee see it, and Klee can perform symbolic execution in a consistent environment.

Symbolic execution generates path constraints that also have to be synchronized. Since QEMU and Klee keep a synchronized CPU, device, and memory state, any write to the state by one of them will be reflected in the path constraints kept by Klee. For example, when symbolically executing driver code accesses concrete kernel memory, it sees data consistent with its own execution so far. Conversely, when concrete code attempts to access a symbolic memory location, that location is automatically concretized, and a corresponding constraint is added to the current path. Data written by concrete code is seen as concrete by symbolically running driver code.

Symbolic Execution of Driver Code

QEMU runs in a loop, continuously fetching guest code blocks, translating them, and running them on the host CPU or in Klee. When a basic block is fetched, DDT checks whether the program counter is inside the driver of interest or not. If yes, QEMU generates a block of LLVM code (or fetches the code from a translation cache) and passes it to Klee; otherwise, it generates x86 machine code and sends it to the host processor.

DDT monitors kernel code execution and parses kernel data structures to detect driver load attempts. DDT catches the execution of the OS code responsible for invoking the load entry point of device drivers. For example, on Windows XP SP3, DDT monitors attempts to execute code at address `0x805A3990`, then parses the stack to fetch the device object. If the name of the driver corresponds to the one being monitored, DDT further parses the corresponding data structures to retrieve the code and data segment locations of the driver. Parsing the data structures is done transparently, by probing the virtual address space, without causing any side effects (e.g., no page faults are induced).

When the driver is executed with symbolic inputs, DDT forks execution paths as it encounters conditional branches. Forking consists primarily of making a copy of the contents of the CPU, the memory, and the devices, to make it possible to resume the execution from that state at a later time. In other words, each execution state consists conceptually of a complete system snapshot.

Optimizing Symbolic Execution

Since symbolic execution can produce large execution trees (exponential in the number of branches), DDT implements various optimizations to handle the large number of states generated by Klee. Moreover, each state is big, consisting of the entire physical memory and of the various devices (such as the contents of the virtual disk).

DDT uses chained copy-on-write: instead of copying the entire state upon an execution fork, DDT creates an empty memory object containing a pointer to the parent object. All subsequent writes place their values in the empty object, while reads that cannot be resolved locally (i.e., do not "hit" in the object) are forwarded up to the parent. Since quick forking can lead to deep state hierarchies, we cache each resolved read in the leaf state with a pointer to the target memory object, in order to avoid traversing long chains of pointers through parent objects.

Symbolic Hardware

For PCI devices, the OS allocates resources (memory, I/O regions, and interrupt line) for the device, as required the device descriptor, prior to loading the driver, and then writes the addresses of allocated resources to the device's registers. From that point, the device continuously monitors all memory accesses on the memory and I/O buses; when an address matches its allocated address range, the device handles the access. In QEMU, such accesses are handled by read/write functions specific to each virtual device. For DDT symbolic devices, the write functions discard their arguments, and the read functions always returns an unconstrained symbolic value. When DDT decides to inject a symbolic interrupt, it calls the corresponding QEMU function to assert the right interrupt assigned to the symbolic device by the OS.

The execution of the driver also depends on certain parts of the device descriptor, not just on the device memory and I/O registers. For example, the descriptor may contain a hardware revision number that triggers slightly different behavior in the driver. Unfortunately, the device descriptor is parsed by the OS when selecting the driver and allocating device resources, so DDT cannot just make it symbolic. Instead, as the device drivers always accesses the descriptor through kernel API functions, we use annotations to insert appropriately constrained symbolic results when the driver reads the descriptor.

Tested Driver Binary File	Size of Driver Code Segment	Size of Driver Source Code	Number of Functions in Driver	Number of Called Kernel Functions	Source Code Available ?
Intel Pro/1000	168 KB	120 KB	525	84	No
Intel Pro/100 (DDK)	70 KB	61 KB	116	67	Yes
Intel 82801AA AC97	39 KB	26 KB	132	32	No
Ensoniq AudioPCI	37 KB	23 KB	216	54	No
AMD PCNet	35 KB	28 KB	78	51	No
RTL8029	18 KB	14 KB	48	37	No

Table 4.1 – Characteristics of Windows drivers used to evaluate DDT.

4.2.2 Fooling the OS into Accepting Symbolic Devices

Hardware buses like PCI and USB support Plug-and-Play, which is a set of mechanisms that modern operating systems use to detect insertion and removal of devices. The bus interface notifies the OS of such events. When the OS detects the presence of a new device, it loads the corresponding driver. The right driver is selected by reading the vendor and device ID of the inserted device. If the driver is for a PCI device, it will typically need to read the rest of the descriptor, i.e., the size of the register space and various I/O ranges.

DDT provides a PCI descriptor for a *fake device* to trick the OS into loading the driver to be tested. The fake device is an empty "shell" consisting of a descriptor containing the vendor and device IDs, as well as resource information. The fake device itself does not implement any logic other than producing symbolic values for read requests. Support for USB is similar: a USB descriptor pointing to a "shell" device is passed to the code implementing the bus, causing the target driver to be loaded.

Hardware descriptors are simple and can be readily obtained. If the actual hardware is available, the descriptors can be read directly from it. If the hardware is not present, it is possible to extract the information from public databases of hardware supported on Linux. If this information is not available, it can be extracted from the driver itself. For example, Windows drivers come with a `.inf` file specifying the vendor and device IDs of the supported devices. The device resources (e.g., memory or interrupt lines) are not directly available in the `.inf` files, but can be inferred after the driver is loaded, by watching for attempts to register the I/O space using OS APIs. We are working on a technique to automatically determine this information directly from the driver.

4.2.3 Exercising Driver Entry Points

DDT must detect that the OS has loaded a driver, determine the driver's entry points, coerce the OS into invoking them, and then symbolically execute them.

DDT automatically detects a driver's entry points by monitoring attempts of the driver to register such entry points with the kernel. Drivers usually export only one entry point, specified

in the driver binary's file header. Upon invocation by the kernel, this routine fills data structures with entry point information and calls a registration function (e.g., `NdisMRegisterMiniport` for network drivers). In a similar way, DDT intercepts the registration of interrupt handlers.

DDT uses Microsoft's Device Path Exerciser as a concrete workload generator to invoke the entry points of the drivers to be tested. Device Path Exerciser is shipped with the Windows Driver Kit [98] and can be configured to invoke the entry points of a driver in various ways, testing both normal and error situations.

Each invoked entry point is symbolically executed by DDT. To accomplish this, DDT returns symbolic values on hardware register reads and, hooks various functions to inject symbolic data. Since execution can fork on branches within the driver, the execution can return to the OS through many different paths. To save memory and time, DDT terminates paths based on user-configurable criteria (e.g., if the entry point returns with a failure).

DDT attempts to maximize driver coverage using pluggable heuristics modules. The default heuristic attempts to maximize basic block coverage, similar to the one used in EXE [20]. It maintains a global counter for each basic block, indicating how many times the block was executed. The heuristic selects for the next execution step the basic block with the smallest value. This avoids states that are stuck, for instance, in polling loops (typical of device drivers). Depending on the driver, it is possible to choose different heuristics dynamically.

DDT tests for concurrency bugs by injecting symbolic interrupts before and after each kernel function called by the driver. It asserts the virtual interrupt line, causing QEMU to interrupt the execution of the current code and to invoke the OS's interrupt handler. The injection of symbolic interrupts is activated as soon as the target driver registers an interrupt handler for the device.

Drivers may legitimately access the kernel's data structures, and this must be taken into account by DDT, to avoid false reports of unauthorized memory accesses. First, drivers access global kernel variables, which must be explicitly imported by the driver; DDT scans the corresponding section of the loaded binary and grants the driver access to them. Second, private kernel data may be accessed via inlined functions (for example, NDIS drivers use macros that access kernel-defined private data fields in the `NDIS_PACKET` data structure). DDT provides annotations for identifying such data structures.

4.3 Discussion

Having seen that DDT is able to automatically find bugs in a reasonable amount of time, we now discuss some of DDT's limitations (§4.3.1) and the tradeoffs involved in testing binary drivers instead of their source code (§4.3.2).

4.3.1 Limitations

DDT subsumes several powerful driver testing tools, but still has limitations, which arise both from our design choices, as well as from technical limitations of the building blocks we use in the DDT prototype.

DDT uses symbolic execution, which is subject to the path explosion problem [15]. In the worst case, the number of states is exponential in the number of covered branches, and this can lead to high memory consumption and long running times for very large drivers. Moreover, solving path constraints at each branch is CPU-intensive. This limits DDT's ability to achieve good coverage for large drivers. We are exploring ways to mitigate this problem by running symbolic execution in parallel [34], and we are developing techniques for trimming the large space of paths to be explored [30]. Any improvements in the scalability of symbolic execution automatically improve DDT's coverage for very large drivers.

This thesis presents a technique that significantly improves the scalability of symbolic execution using state merging (§5). Our current implementation of this technique relies on source-based static analysis and, hence, does not work on closed source device drivers. Extending this technique to work on binaries is a future work (§7).

Like any bug finding tool, DDT might have false negatives. There are two causes for this: not checking for a specific kind of bug, or not covering a path leading to the bug. Since DDT can reuse any existing dynamic bug finding tool (by running it inside the virtual machine along all explored paths) and can be extended with other types of checkers, we expect that DDT can evolve over time into a tool that achieves superior test completeness.

Since DDT does not use real hardware and knows little about its expected behavior, DDT may find bugs that can only be triggered by a malfunctioning device. Even though it has been argued that such cases must be handled in high-reliability drivers [74], for some domains this may be too much overhead. In such cases, these false positives can be weeded out by looking at the execution traces, or by adding device-specific annotations.

Some driver entry points are triggered only when certain conditions hold deep within the execution tree. For example, the `TransferData` entry point in an NDIS driver is typically called when the driver receives a packet *and* provides some look-ahead data from it to the kernel *and* the kernel finds a driver that claims that packet. Since the packet contains purely symbolic data, and is concretized randomly when the kernel reads it, the likelihood of invoking the required handler is low. Annotating the function transmitting the look-ahead data to the kernel can solve this problem.

While testing drivers with DDT can be completely automated, our current DDT prototype requires some manual effort. A developer must provide DDT with PCI device information for the driver's device, install the driver inside a VM, and configure Microsoft Driver Verifier and a concrete workload generator. Once DDT runs, its output is a list of bugs and corresponding

execution traces; the developer can optionally analyze the execution traces to find the cause of the encountered bugs. Even though this limits DDT's immediate usefulness to end users, DDT can be used today by hardware vendors to test drivers before releasing them, by OS vendors to certify drivers, and by system integrators to test final products before deployment.

DDT does not yet support USB, AGP, and PCI-express devices, partly due to the lack of such support in QEMU. This limitation prevents DDT from loading the drivers, but can be overcome by extending QEMU.

Finally, DDT currently has only a 32-bit implementation. This prevents DDT from using more than 4 GB of memory, thus limiting the number of explored paths. Although we implemented various optimizations, like swapping out unnecessary states to disk, memory is eventually exhausted. We ported Klee to 64-bit architectures and contributed it to the Klee mainline; we intend to port DDT as well.

4.3.2 Source-Level vs. Binary-Level Testing

DDT is a binary-level testing tool, and this has both benefits and drawbacks.

A binary tool can test the end result of a complex build tool chain. Device drivers are built with special compilers and linked to specific libraries. A miscompilation (e.g., a wrong field alignment in the data structures), or linking problems (e.g., a wrong library version), can be more easily detected by a binary testing tool.

Binary drivers, however, have meager typing information. The only types used by binaries are integers and pointers, and it may be possible to infer some types by looking at the API functions for which we have parameter information. Nevertheless, it is hard to find type-related bugs that do not result in abnormal operations. For example, a cast of a color value from the framebuffer to the wrong size could result in incorrect colors. Such bugs are more easily detected by source code analyzers.

5 Improving Scalability of Symbolic Execution with Efficient State Merging

As we discuss in subsection 2.3.2, precise symbolic analyses are roughly equivalent in their treatment of loops, but all other design choices (merging at control points, feasibility checking, and function summaries) boil down to choosing which paths to analyze separately vs. which ones to combine into common formulae. In other words, all analyses lie along a spectrum, with search-based symbolic execution (no state merging) at one extreme and whole-program verification condition generation (static state merging) at the other extreme. Instead of making a static design choice of where to be in this spectrum, our approach is to enable a symbolic analysis to choose dynamically the most advantageous point and merge states according to the expected benefit of such a merge.

On the one hand, merging reduces the number of states, but on the other hand, the remaining states become more expensive to explore. In the merged state, each variable that had distinct values in the original states must be constrained by an input-dependent ite expression, which increases the time required to solve future queries involving such variables. If the distinct values were concrete, merging would cause additional solver invocations where expressions could have been evaluated concretely in separate states.

Furthermore, there is an inherent incompatibility between partial searches using coverage-guided search strategies (as is often done in test generation) and static state merging at control flow join points: merging can be maximized by exploring the CFG in topological order, so that a combined state can be computed from its syntactic predecessors. A coverage-guided search strategy, however, will dynamically deprioritize some states (e.g., defer for later the exploration of additional iterations of a loop), which prevents using a topological order.

Therefore, to make state merging practical, we must solve two problems: (1) Automatically identify an advantageous balance between exploring fewer complex states vs. more simpler states, and merge states only when this promises to reduce exploration time; and (2) Efficiently combine state merging with search strategies that deprioritize “non-interesting” execution paths.

To solve the first problem, we developed *query count estimation* (QCE), a way to estimate how variables that are different in two potentially mergeable states will be used in the future. We preprocess the program using a lightweight static analysis to identify how often each variable is used in branch conditions past any given point in the CFG, and use this as a heuristic estimate of how many subsequent solver queries that variable is likely to be part of. Using this heuristic, we check whether two states are sufficiently similar that merging them would yield a net benefit. That is, the additional cost of solving more and harder SMT queries is outweighed by the savings from exploring fewer paths. The results of this static analysis affect only the completion time of the symbolic analysis—not its soundness or completeness.

To solve the second problem, we introduce *dynamic state merging* (DSM), a way to dynamically identify opportunities for merging regardless of the exploration order imposed by the search strategy. Without any restrictions on the search strategy, only states that meet at the same location by chance could ever be merged. To increase the opportunities for merging, we maintain a bounded history of the predecessors of the states in the worklist. When picking the next state to process from the worklist, we check whether some state a_1 is similar to a predecessor a_2' of another state a_2 in the worklist. If yes, then state a_1, which is in some sense lagging behind a_2, is prioritized over the others. This causes it to be temporarily *fast-forwarded*, until its own successor matches up with the candidate-for-merging state a_2. If the state diverges, i.e., one of a_1's successors is no longer sufficiently similar to a predecessor of a_2, the merge attempt is abandoned. Thus, while the search strategy is still in control, DSM identifies merge opportunities dynamically within a fixed distance and only briefly takes over control to attempt the merge. After the merge attempt, the search strategy continues as before. Like QCE, DSM does not affect soundness or completeness of the symbolic analysis.

We combine the solutions to these two problems by using QCE (explained in detail in §5.1) to compute the similarity relation used by DSM (§5.2). Even though we initially developed these techniques to improve the performance of search-based symbolic execution engines for test generation, we believe our analysis and the insights into building efficiently solvable symbolic formulae from programs are applicable to other symbolic program analyses as well.

In the rest of the chapter, we present the query count estimation (§5.1) and the dynamic state merging (§5.2) techniques in detail and discuss our prototype implementation (§5.3).

5.1 Query Count Estimation

We now illustrate the need for estimating the expected benefit of merging using an example (§5.1.1), show how to compute the query count estimates (§5.1.2), and then justify our decisions (§5.1.3).

```
1 void main(int argc, char **argv) {
2   int r = 1, arg = 1;
3   if (arg < argc)
4     if (strcmp(argv[arg], "-n") == 0) {
5       r = 0; ++arg;
6     }
7   for (; arg < argc; ++arg)
8     for (int i = 0; argv[arg][i] != 0; ++i)
9       putchar(argv[arg][i]);
10  if (r)
11    putchar('\n');
12 }
```

Figure 5.1 – Simplified version of the `echo` program.

5.1.1 Motivating Example

Consider the example program in Figure 5.1, a simplified version of the UNIX `echo` utility that prints all its arguments to standard output, except for argument 0, which holds the program name. If the first regular argument is `"-n"`, no newline character is appended. We analyze this program using Algorithm 1, assuming bounded input. Specifically, we assume that `argc` $= N+1$ for some constant $N \geq 1$, and that each of the N command-line arguments, pointed to by the corresponding element of `argv`, is a zero-terminated string of up to L characters. For simplicity, we assume that `strcmp` and `putchar` do not split paths. Under these preconditions, the total number of feasible program paths is $L^N + L^{N-1}$, and the branch condition at line 3 is always true.

The execution paths first split at line 4 on the condition C that $argv[1]$ points to the string "`-n`". Line 6 is then reached by the two states $(6, C, [\texttt{r} = 0, \texttt{arg} = 2])$ and $(6, \neg C, [\texttt{r} = 1, \texttt{arg} = 1])$. These two can be merged into the single (but fully precise) state $(6, \text{true}, [\texttt{r} = \text{ite}(C, 0, 1), \texttt{arg} = \text{ite}(C, 2, 1)])$. Consider now the loop condition `arg` < `argc` in line 7. If the states were kept separate, this condition could be evaluated concretely in both states, as $1 < N+1$ and $2 < N+1$, respectively. In the merged state, however, the condition would become the disjunctive expression $\text{ite}(C, 2, 1) < N+1$, which now requires a solver invocation where it was not previously necessary. The consequences of having merged at line 6 become even worse later in the execution, for the condition at line 8. The array index is no longer concrete, so the SMT solver is required to reason about symbolic memory accesses in the theory of arrays on every iteration of the nested loop. In this example, merging reduces the total number of states, but the merged state is more expensive to reason about. Our experiments confirm that the total time required to fully explore all feasible paths in this program is significantly shorter if the paths are *not* merged on line 6.

Now consider the branching point in the inner loop header at line 8. Since this loop may be executed up to L times, each state that enters the loop creates L successor states, one for each loop exit possibility. For example, a state exiting after the second iteration is $(8, \ldots \wedge \texttt{argv}[1][0] \neq$

$0 \wedge \texttt{argv}[1][1] = 0, [\ldots, \texttt{i} = 1]$). On the next iteration of the outer loop (line 7), each of these L states again spawns L successors. At the end of the N outer loop iterations, there is a total of L^N states. However, all of the states created in the loop at line 8 during the same iteration of the outer loop differ only in the value of the temporary variable `i`, which is never used again in the program. Therefore, merging these states does not increase the cost of subsequent feasibility checks, yet it cuts the number of states after the outer loop down to the number of states before the loop (2 in our example). Note that, while the path condition of the merged state is created as a disjunction, here it can be simplified to the common prefix of all path conditions.

There is another, less obvious, opportunity for merging states. Looking back at the first feasible branch at line 4, consider the state $(7, C, [\texttt{r} = 0, \texttt{arg} = 2])$, which corresponds to the path through the "then" branch, and the state $(7, \neg C, [\texttt{r} = 1, \texttt{arg} = 2])$, which corresponds to the path through the "else" branch and one first iteration over the outer loop. Merging these two states yields the state $(7, \text{true}, [\texttt{r} = \text{ite}(C, 0, 1), \texttt{arg} = 2])$, which introduces a disjunction for the symbolic expression representing the value of the variable `r`. Unlike the `arg` variable we discussed above, `r` is used only once on line 10, just before the program terminates. Therefore, the time saved by exploring the loops at lines 7-9 with fewer states can outweigh the cost of testing the more complex branch condition on line 10 in the merged state.

This example demonstrates that the net benefit of merging two states depends heavily on how often variables whose values differ between two states affect later branch conditions. This is the key insight behind QCE, which we explain next.

5.1.2 Computing the Heuristic

To make an exact merging decision, one would have to compute the cumulative solving times for both the merged and unmerged cases. But this is impractical, so the query count estimation heuristic (QCE) makes several simplifications that allow it to be largely pre-computed before symbolic execution begins. QCE can be calibrated using a number of parameters, which we denote using the Greek letters α, β, and κ.

At each program location ℓ, QCE pre-computes a set $H(\ell)$ of "hot variables" that are likely to cause many queries to the solver if they were to contain symbolic values. The heuristic is to avoid introducing new symbolic values for these "hot variables". Specifically, states should be merged only if every hot variable either has the same concrete value in both states or is already symbolic in at least one of the states. Formally, QCE is implemented by defining the similarity relation $\sim$ of Algorithm 1 as

$$(\ell, pc_1, s_1) \sim_{qce} (\ell, pc_2, s_2) \iff \forall v \in H(\ell) : s_1[v] = s_2[v] \vee I \blacktriangleleft s_1[v] \vee I \blacktriangleleft s_2[v], \tag{5.1}$$

where $I \blacktriangleleft s[v]$ denotes that variable v has a symbolic value in the symbolic store s (i.e., it depends on the set of symbolic inputs I).

In order to check whether a variable v is hot at location ℓ, QCE estimates the number of additional queries $Q_{add}(\ell, v)$ that would be executed after reaching ℓ if variable v were to be made symbolic. Variable v is determined to be hot if this number is larger than a fixed fraction α of the total number of queries $Q_t(\ell)$ that will be executed after reaching ℓ:

$$H(\ell) = \{v \in V \mid Q_{add}(\ell, v) > \alpha \cdot Q_t(\ell)\} \tag{5.2}$$

To estimate these numbers of queries efficiently, we assume that every executed conditional branch leads to a solver query with a fixed probability (which could be taken into account by suitably adjusting the value of α), and that each branch is feasible with a fixed probability β. Consider function q that descends recursively into the control flow graph counting the number of queries that are selected by a function c:

$$q(\ell', c) = \begin{cases} \beta \cdot q(succ(\ell'), c) + \beta \cdot q(\ell'', c) + c(\ell', e) & instr(\ell') = \mathsf{if}\,(e)\,\mathsf{goto}\,\ell'' \\ 0 & instr(\ell') = \mathsf{halt} \\ q(succ(\ell'), c) & \text{otherwise} \end{cases} \tag{5.3}$$

Then $Q_{add}(\ell, v)$ and $Q_t(\ell)$ can be computed recursively as follows:

$$\begin{aligned} Q_{add}(\ell, v) &= q\big(\ell, \lambda(\ell', e).\,\mathrm{ite}\big((\ell, v) \lhd (\ell', e), 1, 0\big)\big) \\ Q_t(\ell) &= q\big(\ell, \lambda(\ell', e).\,1\big), \end{aligned} \tag{5.4}$$

where $(\ell, v) \lhd (\ell', e)$ denotes the fact that expression e at location ℓ' may depend on the value of variable v at location ℓ. For the sake of simplicity, we assume all program loops to be unrolled and all function calls to be inlined. For loops (and recursive function calls) whose number of iterations cannot be determined statically, QCE assumes a fixed maximum number of iterations κ.

Note that the implementation of QCE is limited to estimating the number of additional queries without taking into account the fact that queries may become more expensive due to ite expressions. Our evaluation shows that this suffices in most cases, but we also found a few cases in which lifting this limitation would improve our results. The justification of QCE in §5.1.3 describes how to integrate the cost of ite expressions in the computation.

Interprocedural QCE. In our implementation, we avoid the assumption of inlined functions by computing $Q_{add}(\ell, v)$ and $Q_t(\ell)$ for all function entry points ℓ as function summaries. We do this compositionally, by computing per-function *local* query counts in a bottom-up fashion. The local query counts for a function F include all queries issued inside F and all functions called by F. To compute these, we extend Equation (5.3) to handle function calls. At every call site, the local query counts are incremented by the local query counts at the entry point of the callee. Since the local query counts do not include queries issued after the function returns to the caller (this would require context-sensitive local query counts), we perform the last step of the computation dynamically during symbolic execution. We obtain the global query counts

by adding the local query counts at the location of the current state to the sum of the local query counts of all return locations in the call stack.

Parameters. In our implementation, QCE is parametrized by α, β, and the loop bound κ. Optimal values for these parameters are difficult to compute analytically. For a given program, one can empirically find good parameter values using a simple hill-climbing method. In our experiments, we determined the parameter values this way using four programs and then used these values for the other programs, with good results (see §5.3).

Illustrating Example. Consider again the program in Figure 5.1 with the same input constraints we described in §5.1.1. We now illustrate how QCE can be used to decide whether to merge states at lines 6 and 7. We use the heuristic parameters $\alpha = 0.5$, $\beta = 0.6$ and, to keep the example brief, we set $\kappa = 1$. First, we pre-compute $Q_t(7)$ and $Q_{add}(7, v)$ for $v \in \{\texttt{r}, \texttt{arg}\}$ using Equation (5.4). For brevity, we omit the computation of Q_{add} for `argc`, `argv`, and array contents referenced by `argv`. For $Q_{add}(7, \texttt{arg})$, we get

$$
\begin{aligned}
&Q_{add}(7, \texttt{arg}) = q(7, c)\\
&= \beta q(8, c) + \beta q(10, c) + c(7, \texttt{arg} < \texttt{argc}) = \beta q(8, c) + 1\\
&= \beta(\beta q(9, c) + \beta q(10, c) + c(8, \texttt{argv[arg][i]} \neq 0)) + 1\\
&= \beta(\beta q(9, c) + 1) + 1 = \beta(\beta q(10, c) + 1) + 1 = \beta + 1 = 1.6,
\end{aligned}
$$

where $c = \lambda(\ell', e).\,\text{ite}((7, \texttt{arg}) \lhd (\ell', e), 1, 0)$. Similarly, we compute that $Q_{add}(7, \texttt{r}) = \beta + 2\beta^2 = 1.32$ and $Q_t(7) = 1 + 2\beta + 2\beta^2 = 2.92$ and, according to Equation (5.2), $H(7) = \{\texttt{arg}\}$. As there are no branches between lines 6 and 7, we have $H(6) = H(7) = \{\texttt{arg}\}$. Hence, in this example, the QCE similarity relation (5.1) allows the states at line 6 or 7 to be merged if the values of `arg` in the two states are either equal or symbolic. This is consistent with the results of our manual analysis in §5.1.1.

5.1.3 Justification

We now link our design of QCE to a cost model through the successive application of five key simplifying assumptions. Since QCE is merely a heuristic and not a precise computation, the following only provides a justification for the reasoning behind it, but not a formal derivation. Here we explain a full variant of QCE that includes an estimate of the cost for introducing ite expressions, even though it is not currently implemented in our prototype.

As mentioned above, an optimal heuristic for the similarity relation would compute whether the cumulative solving time T_m for all descendants of the merged state is guaranteed to be less than the combined respective times T_1 and T_2 for the two individual states, i.e., whether $T_m < T_1 + T_2$. In the ideal case of merging two identical states, we would have $T_m = T_1 = T_2$. Thus, merging just two states could theoretically cut the remaining exploration time in half. This is why, in principle, repeated merging can reduce the cumulative solving time by an exponential factor.

Precisely predicting the time required for solving a formula without actually solving it is generally impossible, therefore we apply a first simplification:

Simplifying Assumption 1. *A query takes one time unit to solve. Introducing new* ite *expressions into the query increases the cost to $\zeta > 1$ time units, where ζ is a parameter of the heuristic.*

Thus, we assume the estimated solving time to be linear in the number of queries of each type. In a further simplification, we treat the number of queries that each one of two merge candidates would *individually* cause in the future as equal:

Simplifying Assumption 2. *Two states at the same program location that are candidates for merging will cause the same number Q_t of queries if they are explored separately.*

This simplification is a prerequisite for statically computing query counts for a location in a way that is independent of the actual states during symbolic execution. The merged state then will also invoke these Q_t queries, but some queries will take longer to solve due to introduced ite expressions, and some additional queries become necessary. We denote the number of queries into which merging introduces ite expressions by Q_{ite} (with $Q_{ite} \leq Q_t$). The total cumulative cost of solving these queries is $\zeta \cdot Q_{ite}$, as per our first simplification. Additionally, the merged state can require extra solver invocations for queries corresponding to branch conditions that depend on constant but different values in the individual states (as in the loop conditions on lines 9 and 10 of Figure 5.1). This number of additional queries is Q_{add}.

Note that we ignore the possible cost of introducing disjunctions into the path condition. In many common cases, the different conjuncts of the two path conditions are just negations of each other, and thus the disjunctive path condition can be simplified to the common prefix of the two individual path conditions.

With these simplifications, the total cost of solver queries in the merged state is $1 \cdot (Q_t - Q_{ite})$ for the remaining regular queries plus $\zeta \cdot Q_{ite}$ for queries involving new ite expressions, plus $1 \cdot Q_{add}$ for the additional queries. We can thus formulate the criterion for performing a single merge as $Q_t - Q_{ite} + \zeta \cdot Q_{ite} + Q_{add} < 2 \cdot Q_t$, which simplifies to

$$(\zeta - 1) Q_{ite} + Q_{add} < Q_t. \tag{5.5}$$

The values for Q_t, Q_{ite}, and Q_{add} must be computed over the set of all feasible executions of the merged state. To statically estimate the feasibility of future paths, we add the following simplification:

Simplifying Assumption 3. *Each branch of a conditional statement is feasible with probability $0.5 < \beta < 1$, independently of the other branch.*

We can now estimate the query counts recursively. In the following definition, which is restated from (5.3), function $c(\ell', e)$ can be instantiated for Q_t, Q_{ite}, and Q_{add} individually to return 1 if

checking the feasibility of a branch condition e at location ℓ' causes a query of the specific type (regular, involving ite expressions, or additional), or 0 otherwise:[1]

$$q(\ell', c) = \begin{cases} \beta \cdot q(succ(\ell'), c) + \beta \cdot q(\ell'', c) + c(\ell', e) & instr(\ell') = \text{if}(e)\,\text{goto}\,\ell'' \\ 0 & instr(\ell') = \text{halt} \\ q(succ(\ell'), c) & \text{otherwise} \end{cases} \tag{5.6}$$

For this definition, loop unrolling ensures that conditional statements in loops are counted as many times as the loop can execute. Loops and recursive calls with bounds that are not statically known are unrolled up to a fixed depth, given by the heuristic parameter κ.

The symbolic execution engine issues a query whenever a state (ℓ, pc, s) encounters a branch with a conditional expression e that depends on program input, i.e., e evaluates to an expression $s[e]$ containing variables from the set of inputs I. We denote this by $I \blacktriangleleft s[e]$. To ease notation, we add the following shorthands: we use $s_1[v] \neq_s s_2[v] \overset{def}{\Leftrightarrow} (I \blacktriangleleft s_1[v] \vee I \blacktriangleleft s_2[v]) \wedge s_1[v] \neq s_2[v]$ for the condition causing ite expressions, i.e., symbolic but non-equal variables in two states, and we use $s_1[v] \neq_c s_2[v] \overset{def}{\Leftrightarrow} \neg(I \blacktriangleleft s_1[v] \vee I \blacktriangleleft s_2[v]) \wedge s_1[v] \neq s_2[v]$ for the condition causing additional queries, i.e., concrete and non-equal variables in two states.

To define a function $c(\ell', e)$ for the different types of query counts, we need a method to check whether the branch condition e depends on inputs when reached from one of the individual states. We approximate this statically using a path-insensitive data dependence analysis, and write $(\ell, v) \triangleleft (\ell', e)$ if expression e at location ℓ' may depend on the value of variable v at location ℓ. Thus, we can define the query counts as follows:

$$\begin{aligned} Q_t((\ell, pc_1, s_1), (\ell, pc_2, s_2)) &= q\big(\ell, \lambda(\ell', e). \\ &\qquad \text{ite}(\exists v : (I \blacktriangleleft s_1[v] \vee I \blacktriangleleft s_2[v]) \wedge (\ell, v) \triangleleft (\ell', e)), 1, 0\big) \\ Q_{ite}((\ell, pc_1, s_1), (\ell, pc_2, s_2)) &= q\big(\ell, \lambda(\ell', e). \\ &\qquad \text{ite}(\exists v : s_1[v] \neq_s s_2[v] \wedge (\ell, v) \triangleleft (\ell', e)), 1, 0\big) \\ Q_{add}((\ell, pc_1, s_1), (\ell, pc_2, s_2)) &= q\big(\ell, \lambda(\ell', e). \\ &\qquad \text{ite}(\exists v : s_1[v] \neq_c s_2[v] \wedge (\ell, v) \triangleleft (\ell', e), 1, 0\big) \end{aligned}$$

Computing this recursive relation is expensive, and it cannot be pre-computed before symbolic execution because it requires determining which variables depend on program inputs in the states considered for merging. We therefore assume a fixed probability of input dependence:

Simplifying Assumption 4. *The number of branches whose conditions are dependent on inputs is a fixed fraction φ of the total number of conditional branches.*

[1] Note that, for simplicity of exposition, we only refer to branch conditions here. In practice, other instructions, such as assertion checks or memory accesses with input-dependent offsets, will also trigger solver queries. Our implementation extends the definition of c to account for these queries.

This enables us to eliminate all variable dependencies from Q_t and simplify it to $Q_t(\ell) = \varphi \cdot q\big(\ell, \lambda(\ell', e).1\big)$. Now, Q_t depends only on the program location and can thus be statically pre-computed.

Q_{ite} and Q_{add} count queries for which specific variable pairs are not equal in the two merge candidates. Therefore, we would need to statically pre-compute Q_{ite} and Q_{add} for each subset of variables that could be symbolic in either state during symbolic execution. To eliminate this dependency on the combination of specific variables, we compute query counts for individual variables. The *per-variable query counts* $Q_{ite}(\ell, v)$ and $Q_{add}(\ell, v)$ are defined as the value of $Q_{ite}(\ell)$ and $Q_{add}(\ell)$, respectively, computed as if v was the only variable that differs between the merge candidates. The per-variable query counts can be computed as $Q_{ite}(\ell, v) = Q_{add}(\ell, v) = q\big(\ell, \lambda(\ell', e).\text{ite}\big((\ell, v) \triangleleft (\ell', e), 1, 0\big)\big)$.

Summing the per-variable query counts for all variables that differ between the merge candidates will grossly over-estimate the actual values of Q_{ite} and Q_{add}, since conditional expressions often depend on more than just one variable, and many queries would thus be counted multiple times. Similarly, using just the maximum per-variable query count would cause an under-estimation. In fact,

$$\max_{\{v \in V \mid s_1[v] \neq_c s_2[v]\}} Q_{add}(\ell, v) \leq Q_{add}(\ell) \leq \sum_{\{v \in V \mid s_1[v] \neq_c s_2[v]\}} Q_{add}(\ell, v)$$

and analogously for Q_{ite}. We therefore make a final simplification:

Simplifying Assumption 5. *Total query counts are equal to the maximum per-variable query counts for an individual variable times some factor σ, i.e.,*

$$Q_{ite}(\ell) \approx \sigma \cdot \max_{\{v \in V \mid s_1[v] \neq_s s_2[v]\}} Q_{ite}(\ell, v)$$
$$Q_{add}(\ell) \approx \sigma \cdot \max_{\{v \in V \mid s_1[v] \neq_c s_2[v]\}} Q_{add}(\ell, v).$$

The intuition behind this assumption is that the number of independent variables correlates with the input size and not with the total number of variables. Applying this substitution to Equation (5.5) we can now define the similarity relation $\sim_{qce}$ as

$$(\ell, pc_1, s_1) \sim_{qce} (\ell, pc_2, s_2) \overset{def}{\iff} \tag{5.7}$$
$$(\zeta - 1) \max_{\{v \in V \mid s_1[v] \neq_s s_2[v]\}} Q_{ite}(\ell, v) + \max_{\{v \in V \mid s_1[v] \neq_c s_2[v]\}} Q_{add}(\ell, v) < \frac{Q_t}{\sigma}$$

with

$$Q_{ite}(\ell, v) = Q_{add}(\ell, v) = q\big(\ell, \lambda(\ell', e).\text{ite}\big((\ell, v) \triangleleft (\ell', e), 1, 0\big)\big),$$
$$Q_t(\ell) = \varphi \cdot q\big(\ell, \lambda(\ell', e).1\big),$$

and the recursively descending q as defined in Equation (5.6). For convenience, we rename

$\frac{\varphi}{\sigma}$ to the unified parameter α. Thus, α, β, ζ and the unrolling bound κ remain as the only parameters to QCE. The variant of QCE implemented in our prototype is derived from Equation (5.7) by removing Q_{ite} from the criterion, to arrive at

$$\max_{\{v \in V \mid s_1[v] \neq_c s_2[v]\}} Q_{add}(\ell, v) < \alpha Q_t,$$

which is equivalent to

$$\forall v \in V : s_1[v] \neq_c s_2[v] \rightarrow Q_{add}(\ell, v) < \alpha Q_t.$$

To facilitate an efficient implementation in combination with dynamic state merging, as discussed in the next section, we collect a set of variables that exceed the threshold $H_{add}(\ell) = \{v \in V \mid Q_{add}(v) > \alpha Q_t\}$ and can state the similarity relation as (5.1).

This motivates the use of QCE for estimating the similarity of states. We show that QCE is effective in practice in §6.3.

5.2 Dynamic State Merging

We now explain the challenges for applying state merging in fully automated, precise, but incomplete symbolic program analysis (§5.2.1). To overcome these problems, we motivate (§5.2.2) and introduce (§5.2.3) the dynamic state merging algorithm.

5.2.1 Static Merging and Incomplete Exploration

A symbolic program analysis using static state merging traverses the CFG in topological order and attempts to merge states at every joint point. This allows to perform exhaustive exploration with state merging in the fewest possible steps. To ensure termination, the analysis has to stop loop unrolling at a certain depth, unless loops can be summarized by loop invariants. This method is optimal for verification condition generators that encode full programs with bounded or summarized loops. Search-based symbolic execution engines, however, which typically perform incomplete explorations, do not bound loops but are guided by search strategies that prioritize exploring new code over unrolling additional loop iterations. An exploration in strict topological order would override such strategies and stall the engine by requiring it to fully unroll the possibly infinitely many iterations of a loop before proceeding.

This is a problem even for loops that symbolic execution could, in principle, explore exhaustively. Coverage-oriented search strategies are designed to quickly maximize metrics such as statement coverage. The restriction to topological order interferes with such strategies, reducing their performance or even completely stopping them from achieving any progress towards their goal. We support this argument with experimental evidence in §6.3.3.

Consider the example code in Figure 5.2. Depending on the flag `logPacketHash`, the program

```
1 if (logPacketHash) {
2   hash = computeHash(pkt);
3   log("Packet: %s, hash: %s", pkt->name, hash);
4 } else {
5   log("Packet: %s", pkt->name);
6 }
7 handlePacket(pkt);
```

Figure 5.2 – Example code illustrating how static state merging can interfere with search heuristics.

writes to a log either just the name of a packet or both the name and a hash value. The code then processes the packet. In this example, exploring the "else" branch of the conditional statement on line 1 is fast, while exploring the "then" branch is expensive due to the `computeHash` function processing the entire input packet. A coverage-oriented search strategy will likely choose to explore the "else" branch first to quickly reach `handlePacket`, or switch to the "else" branch after not making progress towards its coverage goal being stuck unrolling loops inside the `computeHash` function. With static state merging, the symbolic execution engine has to merge all states at line 7, which requires all execution paths to be explored exhaustively up to that location. Therefore, no code in the `handlePacket` function can be reached before exploring every path in the `computeHash` function, thus being in conflict with the coverage-oriented search heuristic. This conflict could be solved by allowing the merge of only those states that, according to the chosen search strategy, would reach the same point in a program at about the same time. In our example, the state that takes the "else" branch should not be merged with any state that takes the "then" branch, allowing the search strategy to prioritize it independently.

5.2.2 Rationale Behind Dynamic State Merging

To solve the problems of static state merging, we propose *dynamic state merging* (DSM). DSM does not require states to share the same program location in order to be considered for merging. The rationale behind dynamic state merging is the following: consider two abstract states $a_1 = (\ell_1, pc_1, s_1)$ and $a_2 = (\ell_2, pc_2, s_2)$, with $\ell_1 \neq \ell_2$, that are both in the worklist. Assume that a_1', one of the transitive successors of a_1 (which have not been computed yet) will reach location ℓ_2. Provided that the number of steps required to reach ℓ_2 from a_1 is small, and the expected similarity of a_1' and a_2 is high, enough that merging them will be beneficial, it is worth overriding a coverage-oriented search strategy to compute a_1' next and to then merge it with a_2. We refer to this override as *fast-forwarding*, because a_1 is forwarded to a_2's location with temporary priority before resuming the regular search strategy.

To check whether a_1 can be expected to be similar to a_2 in the near future, we check whether a_1 could have been merged with a predecessor a_2' of a_2, i.e., whether $a_1 \sim a_2'$. The underlying expectation, which our experiments confirm, is that if two states are similar, then their two respective successors after a few steps of execution are also likely (but not guaranteed) to be

Input: Worklist w, choice functions $pickNext_D$ and $pickNext_F$, similarity relation $\sim$, trace function *pred*, threshold δ
Data: Forwarding set F.
Result: The next state to execute.
// Determine the forwarding set
1 $F := \{a \in w \mid \exists a' \in w : \exists a'' \in pred(a', \delta) : a \sim a''\}$;
2 **if** $F \neq \varnothing$ **then** // Choose a state from the forwarding set
3 **return** $pickNext_F(F)$
4 **else** // Choose a state using the driving heuristic
5 **return** $pickNext_D(w)$

Algorithm 2. The *pickNext* method for dynamic state merging.

similar.

Note that fast-forwarding deals with special cases automatically: if a state forks while being fast-forwarded, all children that are still similar to a recent predecessor of a state in the worklist are fast-forwarded. If a state leaves the path taken by the state it is similar to, i.e., fast-forwarding diverges, the state is no longer similar to any predecessor and is thus no longer prioritized.

5.2.3 The Dynamic State Merging Algorithm

The DSM algorithm is an instance of Algorithm 1 with a *pickNext* function as defined in Algorithm 2. DSM relies on an external "driving" heuristic (given as a function $pickNext_D$), such as a traditional coverage-oriented heuristic, to select the next state. However, when the algorithm detects that some states, computed as a set F, are likely to be mergeable after at most δ steps of execution, DSM overrides the driving heuristic and picks the next state to execute from F according to another external search heuristic $pickNext_F$.

The algorithm uses a function $pred(a, \delta)$ to compute the set of predecessors of a within a distance of δ. The function can be defined as $pred(a, \delta) = \{a' \mid \exists n \leq \delta : a \in post^n(a')\}$, where $post(a')$ denotes the set of immediate successor states computed by Algorithm 1 for a state a'. Keeping a precise history of reached states can incur prohibitive space costs, but we reduce the space requirements as follows: states from the history are only used for comparisons with respect to $\sim$, hence it is only required to store the parts of the state that are relevant to the relation. Moreover, if the $\sim$ relation is only sensitive to equality, the implementation can store and compare hash values of the relevant information from past states. In this case, checking whether the state belongs to F is implemented as a simple hash table lookup. Hash collisions do not pose a problem, because a full check of the similarity relation is still performed when fast-forwarding finishes and the states are about to be merged. Moreover, the set F is rebuilt after each execution step, so, if a state was added to F due to a collision, it is unlikely to be added again to F in the next step, as a second collision for two different hash values has low

probability.

The relation $\sim_{qce}$, defined in Equation (5.1) in §5.1.3, can be modified to check for equality only, as required for using hashing in the implementation. We express the condition for variables to be either symbolic or equal in Equation (5.1) by $h(s_1[v]) = h(s_2[v])$, where $h(v) = \text{ite}(I \blacktriangleleft v, \star, v)$ filters out symbolic variables by mapping them to a unique special value. The implementation can thus store just the hash value of $\bigcup_{v \in H(\ell)} h(v)$ for a state. Then $\sim_{qce}$ can be checked by comparing the hash values of the two states (modulo hash collisions).

The function $pickNext_F$ determines the execution order among the states selected for fast-forwarding. In our implementation, we pick the first state from F according to the topological order of the CFG. Thus, states that lie behind with respect to the topological order first catch up and are merged with later states.

5.3 Implementation

We built our prototype on top of the KLEE symbolic execution engine [19]. It takes as input a program in LLVM bitcode [84] and a specification of which program inputs should be marked as symbolic for the analysis: command-line arguments or file contents. KLEE implements precise non-compositional symbolic execution with feasibility checks performed at every conditional branch. It uses search strategies to guide exploration; the stock strategies include random search and a strategy biased toward covering previously unexplored program statements.

To support QCE, we extended KLEE to perform a static analysis of the LLVM bitcode to compute local query count estimates as explained in §5.1.2. The analysis is executed before the path exploration and annotates each program location with the corresponding query count estimates $Q_t(\ell)$ and $Q_{add}(\ell, v)$ as defined in §5.1.2. The pass is implemented as an LLVM per-function bottom-up call graph traversal (with bounded recursion) and performs the analysis compositionally. When analyzing each function, the pass attempts to statically determine trip counts (number of iterations) for loops. If it cannot, it approximates them with the loop bound parameter κ. The QCE analysis tracks the query count for local variables, function arguments, and in-memory variables indexed by a constant offset and pointed to by either a local variable, a function argument, or a global variable. We check data dependencies between variables by traversing the program in SSA form. As LLVM's SSA form handles only local variables, we do not track dependencies between in-memory variables except when loading them to locals. We modified KLEE to compute interprocedural query counts and sets of hot variables $H(\ell)$ dynamically during symbolic execution, following §5.1.2.

We implemented DSM as defined by Algorithm 2 in the form of a search strategy layer in KLEE's stacked strategy system plus an execution tracking system that incrementally computes state hashes (see §5.2.3). Each strategy uses its own logic to select a state from the worklist, but can rely on an underlying strategy whenever it has to make a choice among a set of equally important states. In our case, the DSM strategy returns a state from the fast-forwarding set of

states ($pickNext_F$), or, if this set is empty, it resorts to the underlying driving heuristic to select a state from the general worklist ($pickNext_D$).

6 Evaluation

In this section, we evaluate prototype implementation of all three techniques presented in this thesis. We first present the evaluation results of CPI and CPS (§6.1), then the evaluation results of DDT (§6.2), followed by efficient state merging (§6.3).

6.1 Defeating Control-Flow Hijacks with Code-Pointer Integrity Protection

In this section we evaluate Levee's effectiveness, efficiency, and practicality. We experimentally show that both CPI and CPS are 100% effective on RIPE, the most recent attack benchmark we are aware of (§6.1.1). We evaluate the efficiency of CPI, CPS, and the safe stack on SPEC CPU2006, and find average overheads of 8.4%, 1.9%, and 0% respectively (§6.1.2). To demonstrate practicality, we recompile with CPI/CPS/ safe stack the base FreeBSD plus over 100 packages and report results on several benchmarks (§6.1.3).

We ran our experiments on an Intel Xeon E5-2697 with 24 cores @ 2.7GHz in 64-bit mode with 512GB RAM. The SPEC benchmarks ran on an Ubuntu Precise Pangolin (12.04 LTS), and the FreeBSD benchmarks in a KVM-based VM on this same system.

6.1.1 Effectiveness on the RIPE Benchmark

We described in §3.2 the security guarantees provided by CPI, CPS, and the safe stack based on their design; to experimentally evaluate their effectiveness, we use the RIPE [129] benchmark. This is a program with many different security vulnerabilities and a set of 850 exploits that attempt to perform control-flow hijack attacks on the program using various techniques.

Levee deterministically prevents all attacks, both in CPS and CPI mode; when using only the safe stack, it prevents all stack-based attacks. On vanilla Ubuntu 6.06, which has no built-in defense mechanisms, 833–848 exploits succeed when Levee is not used (some succeed probabilistically, hence the range). On newer systems, fewer exploits succeed, due to built-in

	Safe Stack	CPS	CPI
Average (C/C++)	0.0%	1.9%	8.4%
Median (C/C++)	0.0%	0.4%	0.4%
Maximum (C/C++)	4.1%	17.2%	44.2%
Average (C only)	-0.4%	1.2%	2.9%
Median (C only)	-0.3%	0.5%	0.7%
Maximum (C only)	4.1%	13.3%	16.3%

Table 6.1 – Summary of SPEC CPU2006 performance overheads.

protection mechanisms, changes in the run-time layout, and compatibility issues with the RIPE benchmark. On vanilla Ubuntu 13.10, with all protections (DEP, ASLR, stack cookies) disabled, 197–205 exploits succeed. With all protections enabled, 43–49 succeed. With CPS or CPI, none do.

The RIPE benchmark only evaluates the effectiveness of preventing existing attacks; as we argued in §3.2 and according to the proof outlined in §3.3, CPI renders all (known and unknown) memory corruption-based control-flow hijack attacks impossible.

6.1.2 Efficiency on SPEC CPU2006 Benchmarks

In this section we evaluate the runtime overhead of CPI, CPS, and the safe stack. We report numbers on all SPEC CPU2006 benchmarks written in C and C++ (our prototype does not handle Fortran). The results are summarized in Table 6.1 and presented in detail in Figure 6.1. We also compare Levee to two related approaches, SoftBound [101] and control-flow integrity [1, 137, 135].

CPI performs well for most C benchmarks, however, it can incur higher overhead for programs written in C++. This overhead is caused by abundant use of pointers to C++ objects that contain virtual function tables—such pointers are sensitive for CPI, and so all operations on them are instrumented. Same reason holds for gcc: it embeds function pointers in some of its data structures and then uses pointers to these structures frequently.

The next-most important source of overhead are libc memory manipulation functions, like memset and memcpy. When our static analysis cannot prove that a call to such a function uses as arguments only pointers to non-sensitive data, Levee replaces the call with one to a custom version of an equivalent function that checks the safe pointer store for each updated/copied word, which introduces overhead. We expect to remove some of this overhead using improved static analysis and heuristics.

CPS averages 1.2–1.8% overhead, and exceeds 5% on only two benchmarks, omnetpp and perlbench. The former is due to the large number of virtual function calls occurring at run time, while the latter is caused by a specific way in which perl implements its opcode dispatch: it internally represents a program as a sequence of function pointers to opcode handlers, and its

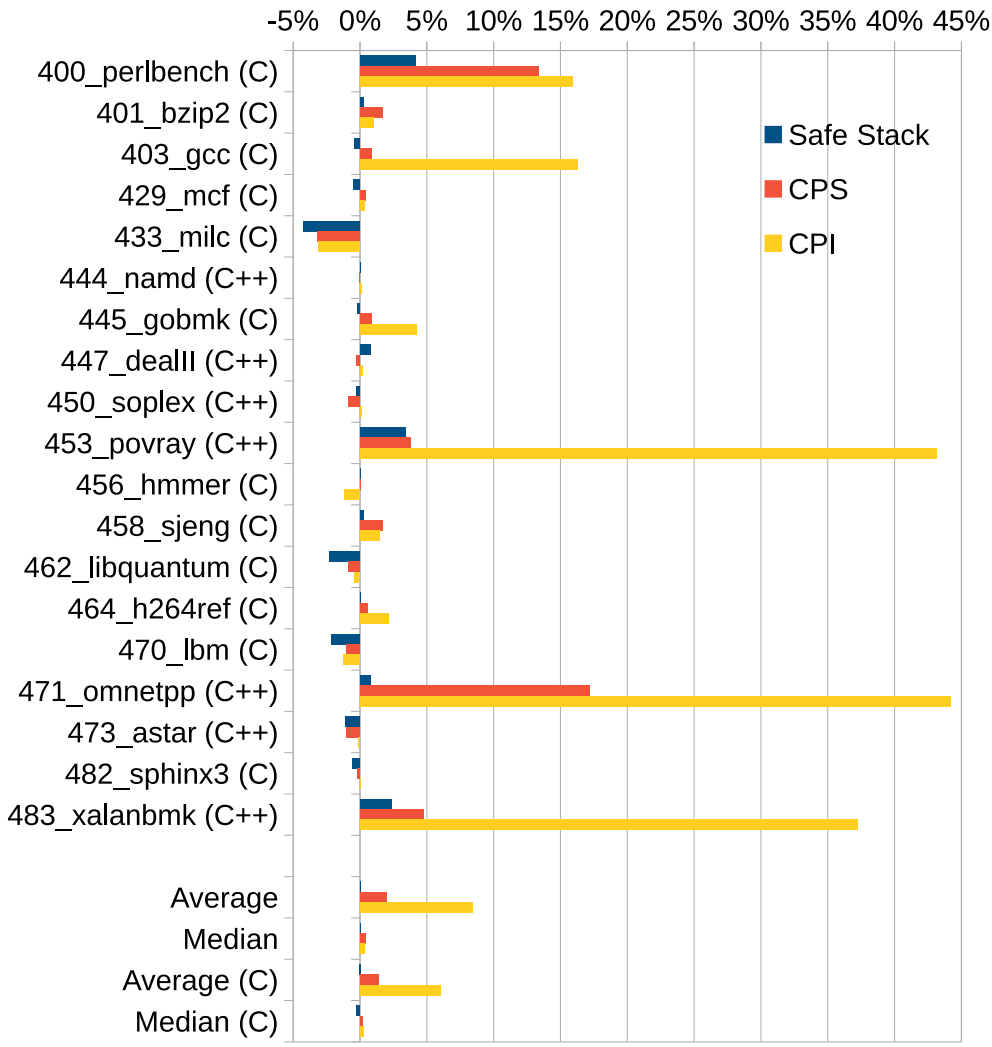


Figure 6.1 – Levee performance for SPEC CPU2006, under three configurations: full CPI, CPS only, and safe stack only.

main execution loop calls these function pointers one after the other. Most other interpreters use a switch for opcode dispatch.

Safe stack provided a surprise: in 9 cases (out of 19), it improves performance instead of hurting it; in one case (namd), the improvement is as high as 4.2%, more than the overhead incurred by CPI and CPS. This is because objects that end up being moved to the regular (unsafe) stack are usually large arrays or variables that are used through multiple stack frames. Moving such objects away from the safe stack increases the locality of frequently accessed values on the stack, such as CPU register values temporarily stored on the stack, return addresses, and small local variables.

The safe stack overhead exceeds 1% in only three cases, perlbench, xalanbmk, and povray. We

studied the disassembly of the most frequently executed functions that use unsafe stack frames in these programs and found that some of the overhead is caused by inefficient handling of the unsafe stack pointer by LLVM's register allocator. Instead of keeping this pointer in a single register and using it as a base for all unsafe stack accesses, the program keeps moving the unsafe stack pointer between different registers and often spills it to the (safe) stack. We believe this can be resolved by making the register allocator algorithm aware of the unsafe stack pointer.

In contrast to the safe stack, stack cookies deployed today have an overhead of up to 5%, and offer strictly weaker protection than our safe stack implementation.

The data structures used for the safe stack and the safe memory region result in memory overhead compared to a program without protection. We measure the memory overhead when using either a simple array or a hash table. For SPEC CPU2006 the median memory overhead for the safe stack is 0.1%; for CPS the overhead is 2.1% for the hash table and 5.6% for the array; and for CPI the overhead is 13.9% for the hash table and 105% for the array. We did not optimize the memory overhead yet and believe it can be improved in future prototypes.

In Table 6.2 we show compilation statistics for Levee. The first column shows that only a small fraction of all functions require an unsafe stack frame, confirming our hypothesis from §3.2.3. The other two columns confirm the key premises behind our approach, namely that CPI requires much less instrumentation than full memory safety, and CPS needs much less instrumentation than CPI. The numbers also correlate with Figure 6.1.

Comparison to SoftBound: We compare with SoftBound [101] on the SPEC benchmarks. We cannot fairly reuse the numbers from [101], because they are based on an older version of SPEC. In Table 6.3 we report numbers for the four C/C++ SPEC benchmarks that can compile with the current version of SoftBound. This comparison confirms our hypothesis that CPI requires significantly lower overhead compared to full memory safety.

Theoretically, CPI suffers from the same compatibility issues (e.g., handling unsafe pointer casts) as pointer-based memory safety. In practice, such issues arise much less frequently for CPI, because CPI instruments much fewer pointers. Many of the SPEC benchmarks either don't compile or terminate with an error when instrumeted by SoftBound, which illustrates the practical impact of this difference.

Comparison to control-flow integrity (CFI): The average overhead for compiler-enforced CFI is 21% for a subset of the SPEC CPU2000 benchmarks [1] and 5-6% for MCFI [106] (without stack pointer integrity). CCFIR [135] reports an overhead of 3.6%, and binCFI [137] reports 8.54% for SPEC CPU2006 to enforce a weak CFI property with globally merged target sets. WIT [3], a source-based mechanism that enforces both CFI and write integrity protection, has 10% overhead[1].

[1] We were unable to find open-source implementations of compiler-based CFI, so we can only compare to published overhead numbers.

Benchmark	FN_{UStack}	MO_{CPS}	MO_{CPI}
400_perlbench	15.0%	1.0%	13.8%
401_bzip2	27.2%	1.3%	1.9%
403_gcc	19.9%	0.3%	6.0%
429_mcf	50.0%	0.5%	0.7%
433_milc	50.9%	0.1%	0.7%
444_namd	75.8%	0.6%	1.1%
445_gobmk	10.3%	0.1%	0.4%
447_dealII	12.3%	6.6%	13.3%
450_soplex	9.5%	4.0%	2.5%
453_povray	26.8%	0.8%	4.7%
456_hmmer	13.6%	0.2%	2.0%
458_sjeng	50.0%	0.1%	0.1%
462_libquantum	28.5%	0.4%	2.3%
464_h264ref	20.5%	1.5%	2.8%
470_lbm	16.6%	0.6%	1.5%
471_omnetpp	6.9%	10.5%	36.6%
473_astar	9.0%	0.1%	3.2%
482_sphinx3	19.7%	0.1%	4.6%
483_xalancbmk	17.5%	17.5%	27.1%

Table 6.2 – Compilation statistics for Levee: FN_{UStack} lists what fraction of functions need an unsafe stack frame; MO_{CPS} and MO_{CPI} show the fraction of memory operations instrumented for CPS and CPI, respectively.

Benchmark	Safe Stack	CPS	CPI	SoftBound
401_bzip2	0.3%	1.2%	2.8%	90.2%
447_dealII	0.8%	-0.2%	3.7%	60.2%
458_sjeng	0.3%	1.8%	2.6%	79.0%
464_h264ref	0.9%	5.5%	5.8%	249.4%

Table 6.3 – Overhead of Levee and SoftBound on SPEC programs that compile and run errors-free with SoftBound.

At less than 2%, CPS has the lowest overhead among all existing CFI solutions, while providing stronger protection guarantees. Also, CPI's overhead is bested only by CCFIR. However, unlike any CFI mechanism, CPI guarantees the impossibility of any control-flow hijack attack based on memory corruptions. In contrast, there exist successful attacks against CFI [61, 41, 22]. While neither of these attacks are possible against CPI by construction, we found that, in practice, neither of them would work against CPS either. We further discuss conceptual differences between CFI and CPI in §2.1.

6.1.3 Case Study: A Safe FreeBSD Distribution

Having shown that Levee is both effective and efficient, we now evaluate the feasibility of using Levee to protect an entire operating system distribution, namely FreeBSD 10. We rebuilt

Benchmark	Safe Stack	CPS	CPI
Static page	1.7%	8.9%	16.9%
Wsgi test page	1.0%	4.0%	15.3%
Dynamic page	1.4%	15.9%	138.8%

Table 6.4 – Throughput benchmark for web server stack (FreeBSD + Apache + SQLite + mod_wsgi + Python +Django).

the base system—base libraries, development tools, and services like bind and openssh—plus more than 100 packages (including apache, postgresql, php, python) in four configurations: CPI, CPS, Safe Stack, and vanilla. FreeBSD 10 uses LLVM/clang as its default compiler, while some core components of Linux (e.g., glibc) cannot be built with clang yet. We integrated the CPI runtime directly into the C library and the threading library. We have not yet ported the runtime to kernel space, so the OS kernel remained uninstrumented.

We evaluated the performance of the system using the Phoronix test suite [108], a widely used comprehensive benchmarking platform for operating systems. We chose the "server" setting and excluded benchmarks marked as unsupported or that do not compile or run on recent FreeBSD versions. All benchmarks that compiled and worked on vanilla FreeBSD also compiled and worked in the CPI, CPS and Safe Stack versions.

Figure 6.2 shows the overhead of CPI, CPS and the safe-stack versions compared to the vanilla version. The results are consistent with the SPEC results presented in §6.1.2. The Phoronix benchmarks exercise large parts of the system and some of them are multi-threaded, which introduces significant variance in the results, especially when run on modern hardware. As Figure 6.2 shows, for many benchmarks the overheads of CPS and the safe stack are within the measurement error.

We also evaluated a realistic usage model of the FreeBSD system as a web server. We installed Mezzanine, a content management system based on Django, which uses Python, SQLite, Apache, and mod_wsgi. We used the Apache ab tool to benchmark the throughput of the web server. The results are summarized in Table 6.4.

The CPI overhead for a dynamic page generated by Python code is much larger then we expected, but consistent with suspiciously high overhead of the pybench benchmark in Figure 6.2. We think it might be caused by the use of some C constructs in the Python interpreter that are not yet handled well by our optimization heuristics, e.g., emulating C++ inheritance in C. We believe the performance might be improved in this case by extending the heuristics to recognize such C constructs.

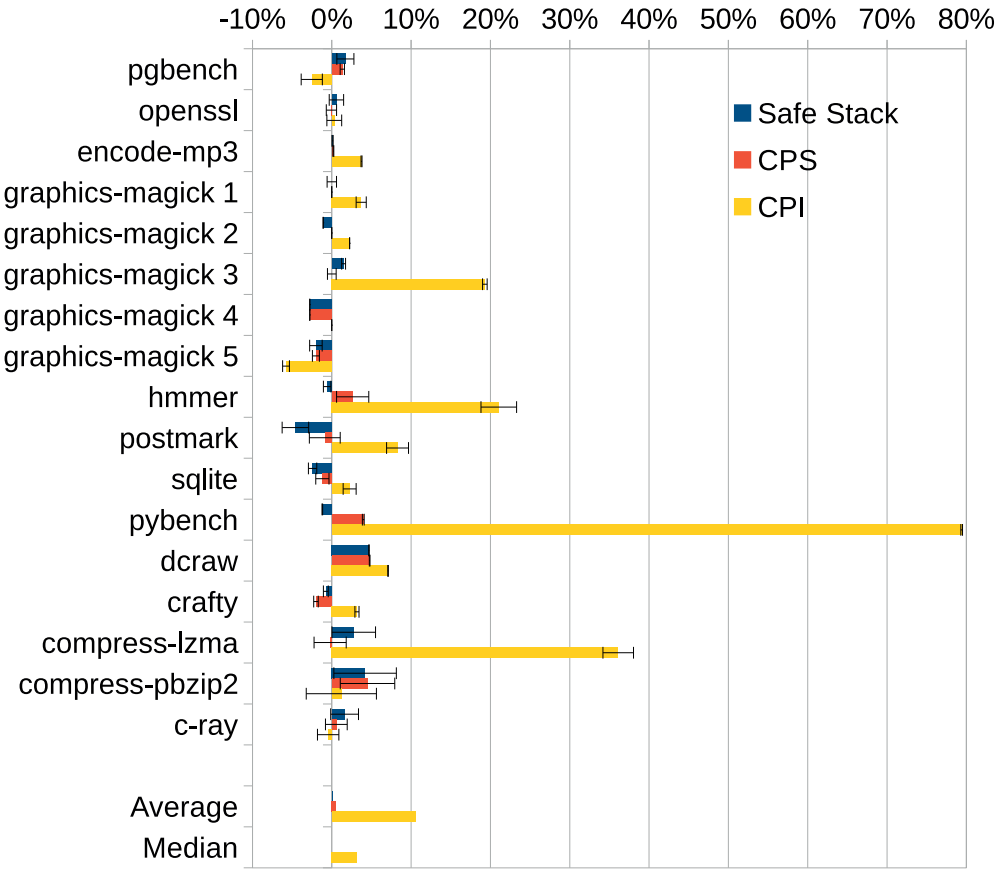


Figure 6.2 – Performance overheads on FreeBSD (Phoronix).

6.2 Automated Device Driver Testing with Selective Symbolic Execution

We applied DDT to six mature Microsoft-certified drivers—DDT found 14 serious bugs (§6.2.1). We also measured code coverage, and found that DDT achieves good coverage within minutes (§6.2.2). All reported measurements were done on an Intel 2 GHz Xeon CPU using 4 GB of RAM.

6.2.1 Effectiveness in Finding Bugs

We used DDT to test four network drivers and two sound card drivers, which use different Windows kernel APIs and are written in both C and C++ (Table 4.1). All drivers are reasonably sized, using tens of API functions; DDT scales well in this regard, mainly due to the fact that it needs no kernel API models. Most of these drivers have been tested by Microsoft as part of the WHQL certification process [98] and have been in use for many years.

DDT found bugs in all drivers we tested: memory leaks, memory corruptions, segmentation

Tested Driver	Bug Type	Description
RTL8029	Resource leak	Driver does not always call `NdisCloseConfiguration` when initialization fails
RTL8029	Memory corruption	Driver does not check the range for `MaximumMulticastList` registry parameter
RTL8029	Race condition	Interrupt arriving before timer initialization leads to BSOD
RTL8029	Segmentation fault	Crash when getting an unexpected OID in `QueryInformation`
RTL8029	Segmentation fault	Crash when getting an unexpected OID in `SetInformation`
AMD PCNet	Resource leak	Driver does not free memory allocated with `NdisAllocateMemoryWithTag`
AMD PCNet	Resource leak	Driver does not free packets and buffers on failed initialization
Ensoniq AudioPCI	Segmentation fault	Driver crashes when `ExAllocatePoolWithTag` returns NULL
Ensoniq AudioPCI	Segmentation fault	Driver crashes when `PcNewInterruptSync` fails
Ensoniq AudioPCI	Race condition	Race condition in the initialization routine
Ensoniq AudioPCI	Race condition	Various race conditions with interrupts while playing audio
Intel Pro/1000	Memory leak	Memory leak on failed initialization
Intel Pro/100 (DDK)	Kernel crash	`KeReleaseSpinLock` called from DPC routine
Intel 82801AA AC97	Race condition	During playback, the interrupt handler can cause a BSOD

Table 6.5 – Summary of previously unknown bugs discovered by DDT.

faults, and race conditions. A summary of these findings is shown in Table 6.5, which shows *all* bug warnings issued by DDT, not just a subset. In particular, we encountered no false positives during testing.

The first two columns of the table are a direct output from DDT. Additionally, DDT produced execution traces that we manually analyzed (as per §4.1.6) in order to produce the last column of the table, explaining each bug. The analyses took a maximum of 20 minutes per bug. Testing each driver took a maximum of 4 hours, and this time includes adding missing API annotations and occasional debugging of the DDT prototype.

From among all bugs found by DDT, only one was related to improper hardware behavior: it was a subtle race condition in the RTL8029 driver, occurring right after the driver registered its interrupt handler, but before it initialized the timer routine and enabled interrupts on the device. If the interrupt fires at this point, the interrupt handler calls a kernel function to which it passes an uninitialized timer descriptor, causing a kernel crash. From the execution traces produced by DDT it was clear that the bug occurred in the driver interrupt handler routine after issuing a symbolic interrupt during driver initialization. We checked the address of the interrupt control register in the device documentation; since the execution traces contained no writes to that register, we concluded that the crash occurred before the driver enabled interrupts.

At the same time, if the device malfunctions and this bug manifests in the field, it is hard to imagine a way in which it could be fixed based on bug reports. It is hard to find this kind of bugs using classic stress-testing tools, even with malfunctioning hardware, because the

interrupt might not be triggered by the hardware at exactly the right moment.

Another interesting bug involved memory corruption after parsing parameters (obtained from the registry) in the RTL8029 driver. The driver does not do any bounds checking when reading the *MaximumMulticastList* parameter during initialization. Later, the value of this parameter is used as an index into a fixed-size array. If the parameter has a large (or negative) value, memory corruption ensues and leads to a subsequent kernel panic. This explanation was easily obtained by looking at the execution traces: a faulty memory read was shown at an address equal to the sum of the base address returned by the memory allocator plus an unconstrained symbolic value injected when reading the registry.

An example of a common kind of bug is the incorrect handling of out-of-memory conditions during driver initialization. In the RTL8029, AMD PCNet, and Intel Pro/1000 drivers, such conditions lead to resource leaks: when memory allocation fails, the drivers do not release all the resources that were already allocated (heap memory, packet buffers, configuration handlers, etc.). In the Ensoniq AudioPCI driver, failed memory allocation leads to a segmentation fault, because the driver checks whether the memory allocation failed, but later uses the returned null pointer on an error handling path, despite the fact that the initial check failed.

An example of incorrectly used kernel API functions is a bug in the Intel Pro/100 driver. In its DPC (deferred procedure call) routine, the driver uses the `NdisReleaseSpinLock` function instead of `NdisDprReleaseSpinLock` (as it should for spinlocks acquired using `NdisDprAcquireSpinLock`). This is specifically prohibited by Microsoft documentation and in certain conditions can lead to setting the IRQ level to the wrong value, resulting in a kernel hang or panic.

We tried to find these bugs with the Microsoft Driver Verifier [98] running the driver concretely, but did not find any of them. Furthermore, since Driver Verifier crashes by default on the first bug found, looking for the next bug would typically require first fixing the found bug. In contrast, DDT finds multiple bugs in one run.

To assess the influence that annotations have on DDT's effectiveness, we re-tested these drivers with all annotations turned off. We managed to reproduce all the race condition bugs, because their detection does not depend on the annotations. We also found the hardware-related bugs, cased by improper checks on hardware registers. However, removing the annotations resulted in decreased code coverage, so we did not find the memory leaks and the segmentation faults.

We initially wanted to compare DDT to the Microsoft SDV tool [10], a state-of-the-art static analysis tool for drivers. Since SDV requires source code, we used the Intel Pro/100 network card driver, whose source code appears in the Windows Drivers Development Kit. Unfortunately, we were not able to test this driver out-of-the-box using SDV, because the driver uses older versions of the NDIS API, that SDV cannot exercise. SDV also requires special entry point annotations in the source code, which were not present in the Intel Pro/100 driver. We resorted to comparing on the sample drivers shipped with SDV itself: SDV found the 8 sample bugs in 12 minutes, while DDT found all of them in 4 minutes.

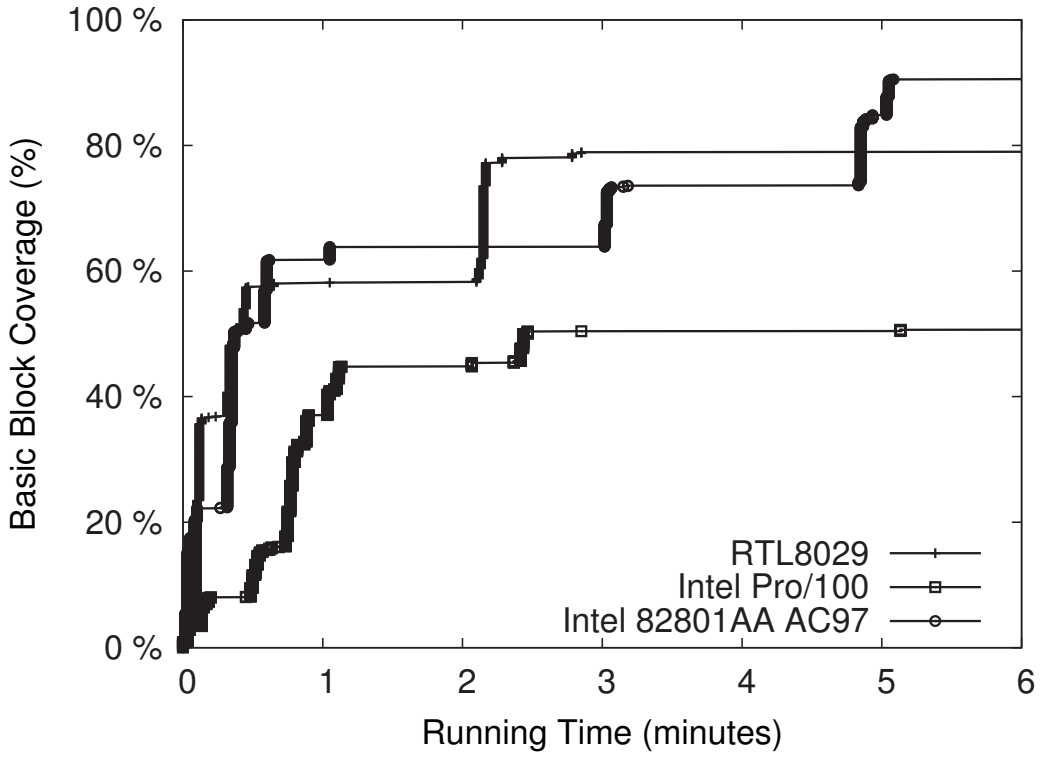


Figure 6.3 – Relative coverage with time

We additionally injected several synthetic bugs in the sample driver (most of these hang the kernel): a deadlock, an out-of-order spinlock release, an extra release of a non-acquired spinlock, a "forgotten" unreleased spinlock, and a kernel call at the wrong IRQ level. SDV did not find the first 3 bugs, it found the last 2, and produced 1 false positive. DDT found all 5 bugs and no false positives in less than a third of the time that SDV ran.

We conclude that DDT can test drivers that existing tools cannot handle, and can find more subtle bugs in mature device drivers. In the next section, we evaluate the efficiency of DDT and assess its scalability.

6.2.2 Efficiency and Scalability

We evaluated DDT on drivers ranging in size from 18 KB to 168 KB. In Figure 6.3 we show how code coverage (as a fraction of total basic blocks) varied with time for a representative subset of the six drivers we tested. In Figure 6.4 we show absolute coverage in terms of number of basic blocks. We ran DDT until no more basic blocks were discovered for some amount of time. In all cases, a small number of minutes were sufficient to find the bugs we reported. For the network drivers, the workload consisted of sending one packet. For the audio drivers, we played a small sound file. DDT's symbolic execution explored paths starting from the exercised entry points. For more complex drivers, workload can be generated with the Device

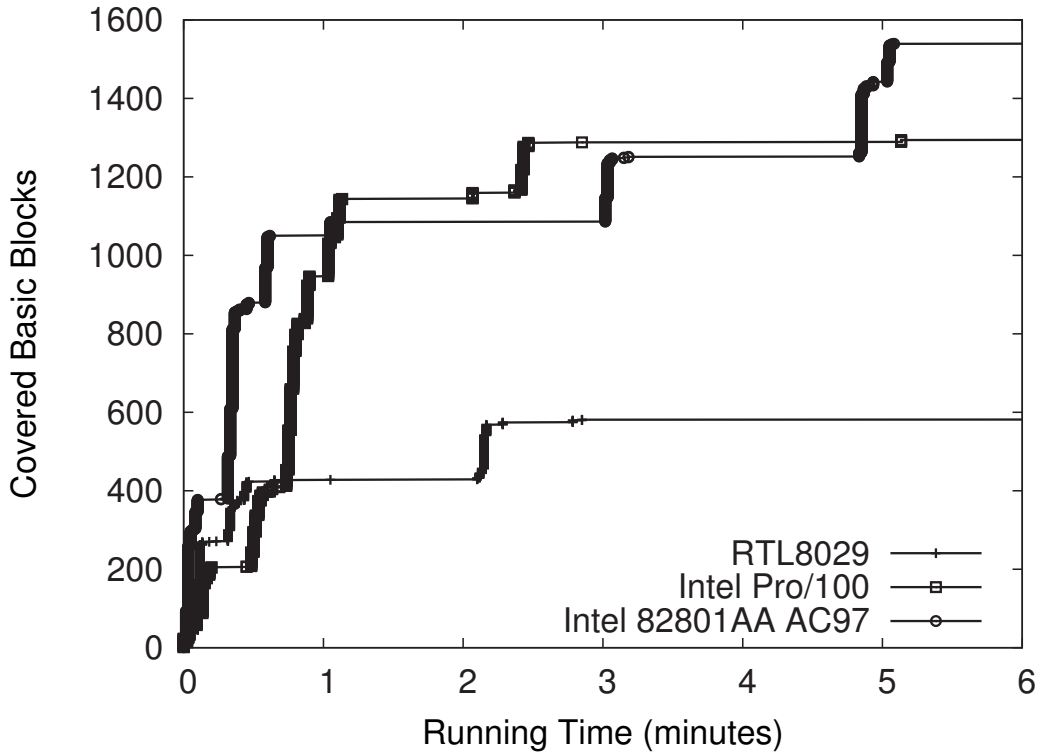


Figure 6.4 – Absolute coverage with time

Path Exerciser (described in §4.2).

DDT has reasonable memory requirements. While testing the drivers in Table 4.1, DDT used at most 4 GB of memory, which is the current prototype's upper limit.

The coverage graphs show long flat periods of execution during which no new basic blocks are covered. These periods are delimited by the invocation of new entry points. The explanation is that the driver-loading phase triggers the execution of many new basic blocks, resulting in a first step. Then, more paths are exercised in it, without covering new blocks. Finally, the execution moves to another entry point, and so on. Eventually, no new entry points are exercised, and the curves flatten.

Overall, the results show that high coverage of binary drivers can be achieved automatically in just a few minutes. This suggests that DDT can be productively used even by end users on their home machines.

6.3 Improving Scalability of Symbolic Execution with Efficient State Merging

We now show that, in practice, our approach attains exponential speedup compared to base symbolic execution (Figure 6.7). We first present our evaluation metrics (§6.3). We then evaluate how DSM combined with QCE improves the thoroughness of program exploration in a time-bounded scenario (§6.3.1). We then show that QCE lies at a sweet spot between single-path exploration and static merging (§6.3.2). Finally, we demonstrate that DSM is essential for combining state merging with coverage-oriented search strategies in incomplete exploration (§6.3.3).

Evaluation Targets. We performed all our experiments on the COREUTILS suite of widely used UNIX command-line utilities, ranging from file manipulation (`cp`, `mv`, etc.) to text processing (`cut`, `sort`, etc.) and shell control flow (e.g., `test`). The total size of the COREUTILS code is 72.1 KLOC, as measured by SLOCCOUNT [128]. We tested these tools using symbolic command line arguments and `stdin` as input. In some cases, the symbolic input size is small enough for KLEE to complete the exploration in less than 5 minutes. We discarded these data points from our evaluation, since such a short period of time is dominated by the constant overhead of our static analysis, whereas we are interested in evaluating the prototype's asymptotic behavior.

Depending on the purpose of each experiment, we employ different driving heuristics: for complete explorations, we used random search, while for partial explorations aimed at obtaining statement coverage, we employed the coverage-oriented search heuristics in [19].

Evaluation Metrics. We evaluate our prototype by comparing it to non-merging search-based symbolic execution as implemented in KLEE. We perform the comparison according to (1) the *amount of exploration* performed given a fixed time budget, and (2) the *time* necessary to complete a fixed exploration task, i.e., the exhaustive search of a set of paths determined by a given symbolic input.

Estimating Number of Paths in Merged States. A direct comparison of the amount of exploration between merge-based and regular symbolic execution is difficult, because counting the feasible paths that have been explored with state merging requires checking the feasibility of each path individually. This is as hard as repeating the exploration without merging, so it is impractical in the cases where symbolic execution without merging times out.

We therefore estimate the path count in a merged state with the help of *state multiplicity*: the multiplicity of a single-path state is 1; when two states merge, the multiplicity of the resulting state is the sum of their multiplicities. State multiplicity over-estimates the path count because, when the state is split at later branches, state multiplicity carries over to both child states, effectively doubling the counted number of paths at each branch (assuming that, as long as a branch is feasible for the merged state, it is also feasible for all the paths represented by it).

For our estimation, we made the assumption that "path explosion" can be modeled as an

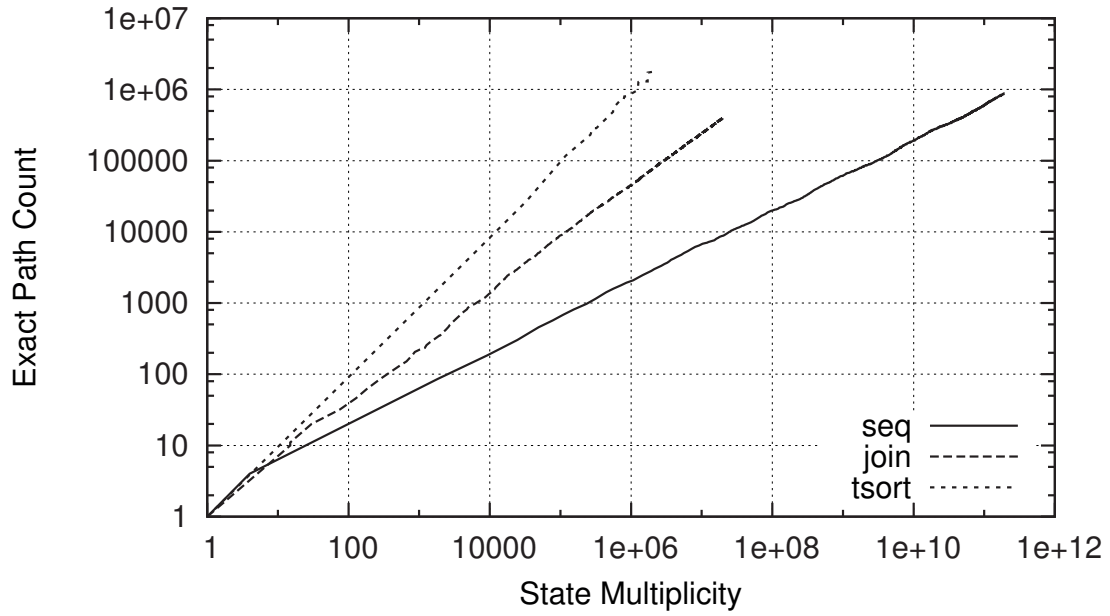


Figure 6.5 – The exact number of paths as a function of state multiplicity for 3 COREUTILS tools. Both axes are logarithmic.

exponential function $a \cdot 2^{b \cdot n}$, where a and b are program-specific constants, and the growth parameter n indicates the progress of unrolling the program CFG along the paths of interest. Both the path count p and state multiplicity m are then based on this formula, each with different values for the constants a and b. Since n depends on the behavior of the search strategy, it cannot be easily determined. However, we can avoid computing n because, at any point in time, both the exact path count and the state multiplicity have the same growth parameter n corresponding to the current exploration progress. Hence, we can take the two model equations $m = a_m \cdot 2^{b_m \cdot n}$ and $p = a_p \cdot 2^{b_p \cdot n}$ and obtain n from the first equation and substitute it in the second. This yields a relation between p and m of the form $\log p \approx c_1 + c_2 \log m$, where c_1 and c_2 are program-dependent coefficients.

To empirically validate this relation, we extended our prototype to accurately track the number of feasible paths by maintaining all the original single-path states along with the merged states. We counted the exact number of paths and the state multiplicity for 1 hour and confirmed a linear relation between the logarithms of the two values. Figure 6.5 illustrates the dependency between m and p with representative measurements of c_2 for 3 COREUTILS.

In the experiments reported in the rest of this section, we approximated the number of paths for state merging as follows. First, we ran the experiments for 1 hour, while accurately tracking the number of feasible paths as explained above (if exploration with merging was 1000× faster, this 1 hour would correspond to ~4 seconds of exploration). From this data, we computed the values of c_1 and c_2. Second, we ran the full experiments, while tracking only the state multiplicity for merged states, which does not incur a significant overhead. Using c_1 and c_2, we then converted the state multiplicity values into the estimation of the feasible path count.

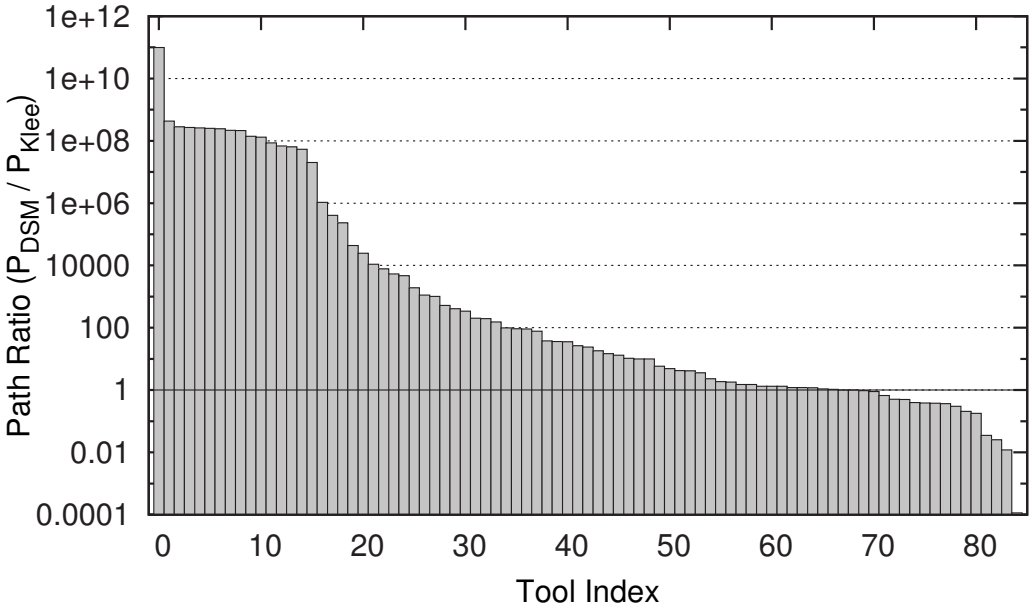


Figure 6.6 – Relative increase in explored paths for DSM + QCE vs. regular KLEE (1h time budget). Each bar represents a COREUTIL.

6.3.1 Faster Path Exploration with DSM and QCE

We were first concerned with how much DSM and QCE, when combined, speed up symbolic execution. We let both our prototype and KLEE run for 1 hour on each of the COREUTILS, and we measured the number of paths explored by each tool. The size of the symbolic inputs passed to each utility was large enough to keep each tool busy for the duration of each run.

Figure 6.6 shows, for each of the tested COREUTILS, a bar representing the ratio between the number of program paths explored by our prototype and KLEE, respectively. The results indicate that our technique explores up to 11 orders of magnitude more paths than plain symbolic execution in the same amount of time. On 14 utilities, our prototype explored fewer paths, which we believe to be due to ignoring ite expressions (see §5.1.3) and limitations of our prototype (see §5.3). Our prototype crashed on 5 utilities, which we do not include in our results. We show a single representative for every tool aliased by multiple names (e.g., `ls`, `dir`, and `vdir`).

6.3.2 Achieving Exponential Speedup with QCE

We now answer the question of how much faster can our technique exhaustively explore a program for a fixed input size. In exhaustive exploration, a coverage-oriented search strategy is not necessary, therefore we focused on the effects of QCE alone. We used it in implementing a selective form of static state merging instead of DSM. Static state merging (SSM) chooses states from the worklist in topological order and attempts to merge them at control flow join points.

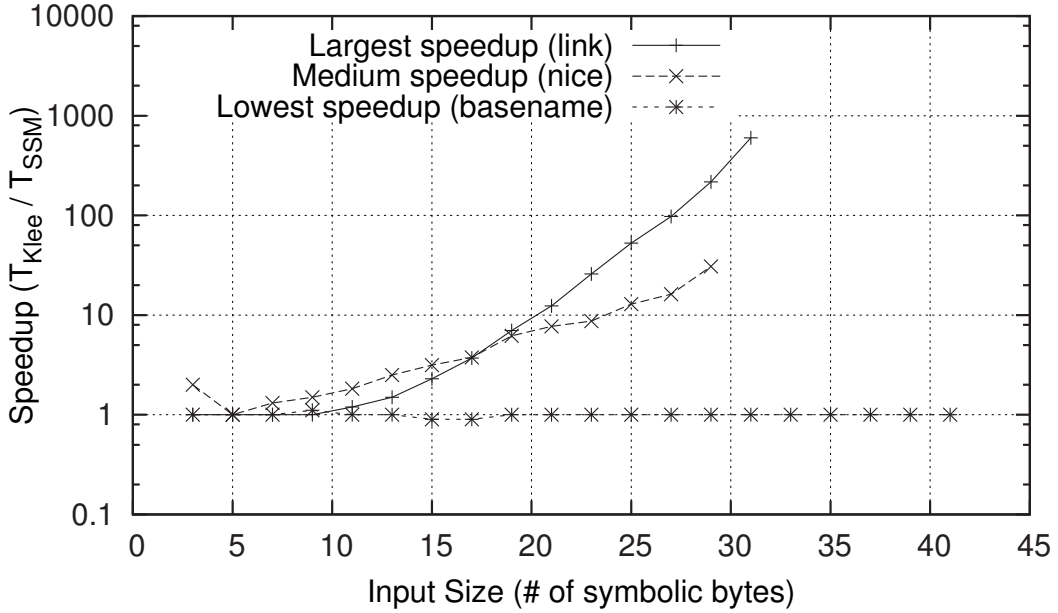


Figure 6.7 – Speedup of QCE versus input size for exhaustive exploration of three representative COREUTILS.

Selective SSM uses query count estimation to keep some states separate. Intuitively (and confirmed experimentally in §6.3.3), SSM performs better than DSM in exhaustive exploration, since it avoids unnecessary computation. However, for incomplete exploration, SSM performs worse than DSM.

We evaluate QCE from two perspectives. First, we look at how much QCE speeds up state merging for complete path exploration tasks. Second, we look at how the QCE heuristic parameters influence performance. We collected our measurements by running each of the COREUTILS for 2 hours, using plain KLEE, and QCE with SSM. For each configuration, we ran experiments with multiple values of symbolic input size, and we measured the corresponding completion time.

Exhaustive Exploration. Figure 6.7 shows the evolution of the completion time ratio (speedup) between SSM using QCE and plain KLEE for three representative COREUTILS programs, as we increase the symbolic input size. One of them achieved the highest speedup, another shows an average speedup, while the last does not show any improvement. This graph illustrates that our prototype completed the exploration goal exponentially faster as the symbolic input size increased. Figure 6.7 also shows that applying QCE does not always lead to speedup. However, Figure 6.8 shows this is actually an infrequent case. The scatter plot illustrates how the execution time of SSM using QCE compares to KLEE when aggregating over the entire set of experiments. The black dots correspond to experiment instances where both our prototype and KLEE finished on time, while the triangles on the right side correspond to situations where KLEE timed out after 2 hours (and thus indicate a lower bound on the actual speedup). The gray disks indicate the relative size of the symbolic inputs used in each experiment instance.

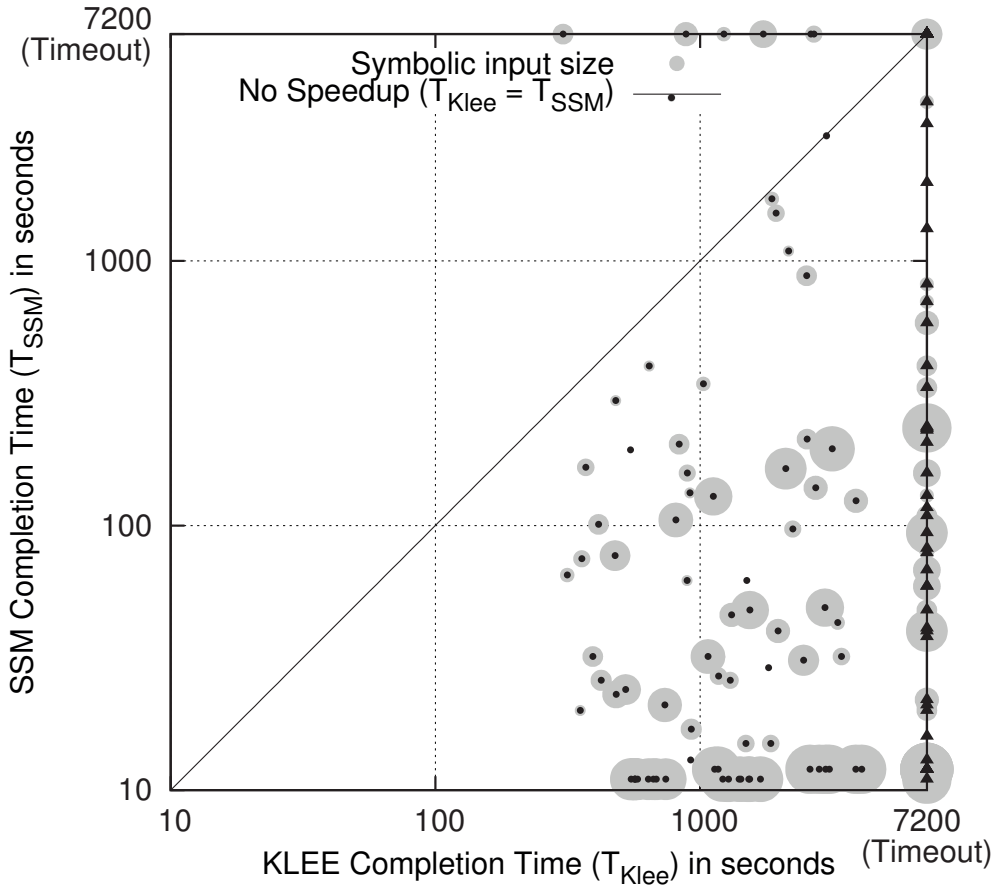


Figure 6.8 – QCE + SSM vs. plain KLEE with varying input sizes (shown as gray disk size). Triangles denote that KLEE timed out after 2h and are thus *lower* bounds on the actual speedup.

Since, in the majority of cases, KLEE times out without completing the exploration, we only show on this graph the timeout points of the smallest input size for each tool, which give a loose lower bound on the speedup. We notice that most of the points in Figure 6.8 are located in the lower-right part of the graph, which correspond to higher speedup values. Moreover, in the cases where both KLEE and our prototype finish on time, large speedups (the lower-right part of the graph) tend to correspond to larger sizes of the symbolic input (larger gray circles). This re-confirms the fact that our speedup is proportional to the size of program input, i.e., the very source of the exponential growth of paths that bottlenecks KLEE.

To get a better understanding of the behavior of QCE, we took a deeper look at the execution of several COREUTILS tools: some that exhibit high speedup, and some for which the performance of our prototype is worse than KLEE's. We extended our prototype to explore states in both merged and unmerged forms. For every solver query in a merged state, we matched all the queries during the same execution step in all the corresponding unmerged states. We then compared query times in the merged and unmerged versions and identified those that became slower due to state merging.

We discovered that, even for tools that exhibited high overall speedup, some queries are more expensive in the merged state than the corresponding queries for the unmerged states combined. In these cases, however, the slowdown was amortized by the reduction in the total number of states to explore, and hence the total number of queries to solve. A typical example is the `sleep` utility, which reads a list of integers from the command line and sums their value in the variable `seconds`. It then validates the resulting value and performs the actual sleep operation. Here, QCE does not identify `seconds` as a hot variable, and all states forked during parsing are merged into a single state, avoiding the exponential increase in paths for each additional integer parsed. Since the value of `seconds` depends on the parsing result, it becomes a complex symbolic expression in the merged state, leading to several complex solver queries in the validation code. Nevertheless, these queries are amortized by the substantial reduction in the number of states to analyze. This example shows that QCE does allow merging of states that differ in live variables, so it is strictly more general than methods based on live variable analysis [15].

In cases where our prototype performed worse than plain KLEE, we observed a large number of queries that were more expensive in the merged state. These queries commonly contained ite expressions and disjunctive path conditions introduced by state merging. The former case shows that our QCE prototype can be improved by including the estimation of ite expressions introduced by state merging, as described in §5.1.3. The latter suggests using a constraint solver that can handle disjunctions more efficiently.

Influence of Heuristic Parameters. The values of the QCE parameters α, β, and κ affect the exploration time of each program. We determined optimal values for α and β experimentally, using hill-climbing over four COREUTILS chosen at random, and obtained $\alpha = 10^{-12}$ and $\beta = 0.8$. We then reused the same values for all other experiments in our evaluation and found that these values perform well in practice. Regarding the loop bound κ, we noticed that many of the loops with an input-dependent number of iterations actually iterate over program inputs. Hence, we chose $\kappa = 10$ corresponding to the average input size in our experiments.

We observed that, among these parameters, the value of α has the highest impact on the running time. In essence, α controls how aggressively the engine tries to merge. When α is ∞, no variables are determined to be hot, and QCE allows all states to be merged. When α is 0, states that contain variables with different concrete values are never merged. Due to this property, we call α the QCE *threshold parameter*. To illustrate this dependency, we randomly chose four COREUTILS (`link`, `nice`, `paste`, `pr`) and ran them using SSM for up to 1 hour with different values of α. Figure 6.9 shows, for each target program, the dependency of the completion time on the threshold parameter. The special point "(no merge)" on the x-axis corresponds to executions with state merging disabled. Note that here we did not cut execution times below 5 minutes.

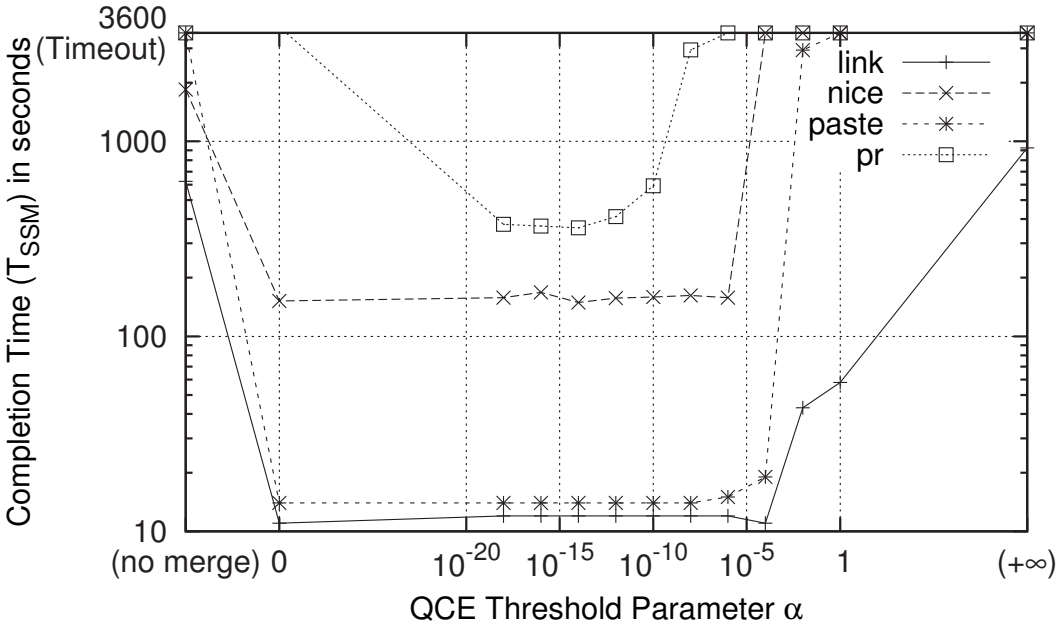


Figure 6.9 – Impact on performance of the threshold parameter α.

6.3.3 Reaching an Exploration Goal with DSM

We now verify whether DSM allows the underlying driving heuristic to reach the goal while still merging states according to QCE. To isolate the effects of DSM, we compare DSM to our SSM implementation, with both using QCE in making merge decisions.

First, we use the coverage-oriented search heuristic from [19] with DSM, and look at how much statement coverage it can achieve in an incomplete setting (1 hour timeout and large search space). The value of the fast-forwarding distance δ was chosen experimentally to equal 8 basic blocks. Figure 6.10 compares the increase in statement coverage obtained by DSM and SSM over base KLEE on those COREUTILS for which the exploration remained incomplete after 1 hour. SSM consistently obtains worse coverage values, confirming its inability to adapt to the exploration goal. However, DSM roughly matches the coverage values of the underlying driving heuristic. Thus, the experiment confirms that DSM's merging avoids interfering with the logic of the driving heuristic, while traversing orders of magnitude more paths (§6.3.1). Even though these additional paths do not necessarily increase statement coverage, they can increase other coverage metrics and ultimately offer higher confidence in the resulting tests. We measured that, on average, 69% of the states selected for fast-forwarding were successfully merged with another state. Hence, the DSM approach to predict state similarity (§5.2.3) works well in practice.

Second, we evaluated the penalty of DSM compared to SSM in *exhaustive* exploration (see §6.3.2). For this experiment, we ran both techniques with varying input sizes. Figure 6.11 aggregates the results in a scatter plot. Most data points are grouped around the diagonal, indicating that the performance of both techniques is comparable, even though DSM is slower

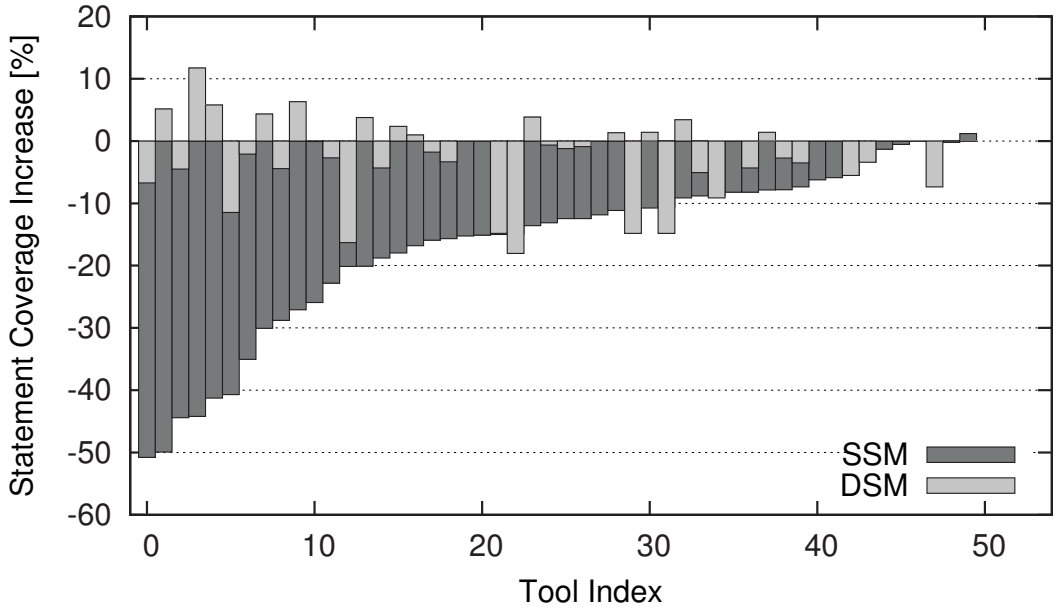


Figure 6.10 – Change in statement coverage of DSM and SSM vs. regular KLEE for a coverage-oriented, incomplete exploration.

than SSM by 15% on average.

We conclude that DSM, while being slightly less efficient than SSM in exhaustive exploration, meets its purpose of allowing the driving heuristic to follow the exploration goal.

Overall, the combination of DSM and QCE offers exponential speedups over KLEE, which suggests that these are two important steps towards improving the performance of symbolic execution.

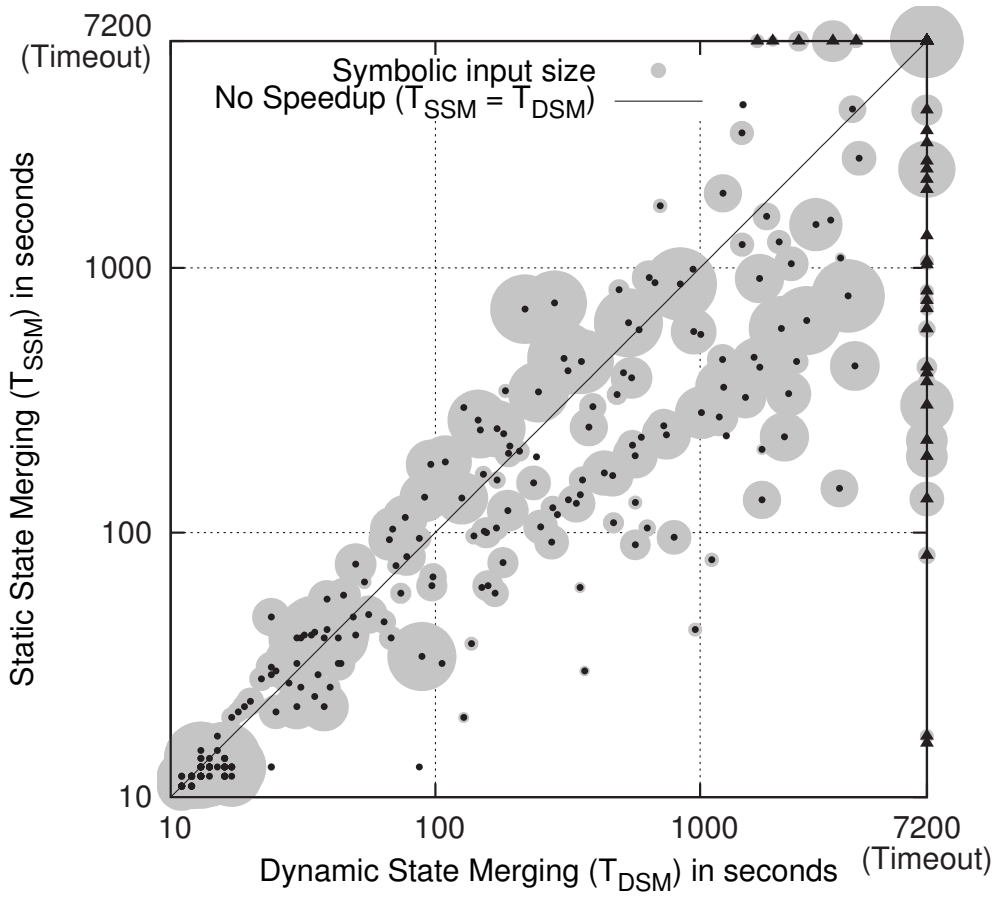


Figure 6.11 – Comparison between the time needed to achieve exhaustive exploration for SSM and DSM.

7 Future Work

This chapter discusses our ongoing and future work aimed to further improve techniques that strengthen the security and reliability of software systems without affecting the system performance or the productivity of software developers.

The key remaining challenge in our quest to build an OS distribution that is fully protected against control-flow hijack attacks by CPI is to apply CPI to the OS kernel. One of the main challenges in doing so is enforcing the code integrity assumption required by our threat model. In the user space, this assumption is enforced by making all code pages read-only and all writeable pages non-executable. However, it is not sufficient in the kernel space: inside the kernel, the attacker can directly modify page protection flags using compromised memory writes. We plan to resolve this challenge by mediating all accesses to the page table through a lightweight hypervisor, as proposed by Dautenhahn at al. [40]. Other challenges of applying CPI to the OS kernel include supporting assembly code, interrupts and DMA.

The average overhead of CPI for C++ programs is significantly higher than for programs written in C. This is because, due to the way C++ implements virtual function calls, every pointer to an object with virtual functions becomes sensitive and has to be protected by CPI. At the same time, the semantics of C++ restricts potential targets of virtual function calls stricter than other indirect control flow transfers, so merely enforcing this semantics provides a strong form of control-flow integrity protection for virtual function calls with low performance overhead [124]. This opens a possibility to combine such enforcement to protect virtual table pointers with CPI to protect all other sensitive pointers in a program. Such combination could offer stronger protection for C++ programs than CPS or CFI at a lower overhead than CPI.

We presented a formal model of CPI along with its correctness proof for that formal model (§3.3). In our future work, we plan to prove correctness of the actual CPI implementations using the Vellvm framework [138].

We implemented and evaluated the query count estimation (QCE) heuristic for efficient state merging in the context of symbolic execution, however, as we discuss in §2.3.2, it can be

applied to other flavors of precise symbolic analysis as well. In our future work, we plan to implement and evaluate QCE for bounded model checking and other verification condition generation approaches (e.g., [35, 72, 130, 9]).

The QCE heuristic relies on source-based static analysis, which prevents tools designed to analyze binaries, such as S^2E and DDT, from taking advantage of the benefits the efficient state merging provides. Although it is possible to convert binaries to an intermediate representation form that is accepted by QCE [28, 46, 6], the output of such conversion lacks high-level concepts, such as variables and types, which are necessary for QCE to be effective. In our future work, we plan to device a version of the QCE heuristics specifically tailored to work on binaries and use it to improve the performance of S^2E and DDT.

Our experimental results demonstrate that QCE is sufficiently precise to ensure significant increase is the net performance of symbolic execution with state merging. However, the approximations behind QCE (§5.1.3) inherently limit the top speedup it can provide. An alternative approach is to optimistically merge states whenever possible, and then dynamically un-merge the states whose analysis speed is negatively affected by the ite expressions created during state merging. This approach does not rely on static analysis and, hence, works equally well on source code or binaries. Our initial experiments on dynamic un-merging in DDT demonstrate promising results.

8 Conclusion

We presented three complementary approaches to strengthen reliability and security of systems software without affecting the system performance or the productivity of software developers.

First, we described *code-pointer integrity* (CPI), a way to protect systems against all control-flow hijacks that exploit memory bugs, and *code-pointer separation*, a relaxed form of CPI that still provides strong guarantees. The key idea is to selectively provide full memory safety for just a subset of a program's pointers, namely code pointers. We implemented our approach and showed that it is effective, efficient, and practical. Given its advantageous security-to-overhead ratio, we believe our approach marks a step toward deterministically secure systems that are fully immune to control-flow hijack attacks.

Second, we presented *DDT*, a tool for testing closed-source binary device drivers against undesired behaviors, like race conditions, memory errors, and resource leaks. We evaluated *DDT* on six mature Windows drivers and found 14 serious bugs that can cause a system to freeze or crash.

DDT combines virtualization with selective symbolic execution to thoroughly exercise tested drivers. A set of modular dynamic checkers identify bug conditions and produce detailed, executable traces for every path that leads to a failure. We showed how these traces can be used to provide evidence of the found bugs, as well as help understand and fix them.

DDT does not require access to source code and needs no assistance from users, thus making it widely applicable. We envision *DDT* being used by IT staff responsible for the reliability and security of desktops and servers, by OS vendors and system integrators, as well as by consumers who wish to avoid running buggy drivers in their operating system kernels.

Finally, we presented efficient state merging technique that improves scalability of symbolic execution. In symbolic execution, state merging reduces the number of states that have to be explored, but increases the burden on the constraint solver. We introduced two techniques for reaping practical benefits from state merging: query count estimation and *dynamic state*

merging. With this combination of techniques, state merging becomes completely dynamic and benefit-driven, unlike static strategies such as static merging or precise function summaries. We experimentally confirmed that our approach can significantly improve exploration time and coverage. This suggests that we have indeed come close to a sweet spot in balancing the simplicity of exploring single paths vs. the reduction of redundancy in exploring multiple paths in a merged state.

Other types of precise symbolic program analysis face similar design choices for grouping paths, but approach the sweet spot from a different angle. Therefore, we believe that our results generalize beyond just symbolic execution and that, for example, query count estimation can serve as a partitioning strategy for verification conditions in bounded model checking.

The three techniques presented in this thesis complement each other: code-pointer integrity increases the security of systems software, DDT empowers end-users to evaluate the security and reliability of systems software they use, and efficient state merging improves the scalability of bug finding tools like DDT.

Bibliography

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *ACM Conf. on Computer and Communication Security*, 2005.

[2] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, 2010.

[3] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *IEEE Symp. on Security and Privacy*, May 2008.

[4] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy Bounds Checking: An Efficient and Backwards-compatible Defense Against Out-of-bounds Errors. In *USENIX Security Symposium*, 2009.

[5] Gautam Altekar and Ion Stoica. Focus replay debugging effort on the control plane. *USENIX Workshop on Hot Topics in Dependability*, 2010.

[6] Kapil Anand, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. A compiler-level intermediate representation based binary analysis and rewriting system. In *ACM EuroSys European Conf. on Computer Systems (EUROSYS)*, 2013.

[7] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. 2008.

[8] Ariane 5 flight 501 failure, report by the inquiry board. http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html, 1996.

[9] Domagoj Babic and Alan J. Hu. Calysto: scalable and precise extended static checking. 2008.

[10] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In *ACM EuroSys European Conf. on Computer Systems (EUROSYS)*, 2006.

[11] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conf. (USENIX)*, 2005.

[12] Sandeep Bhatkar, Eep Bhatkar, R. Sekar, and Daniel C. Duvarney. Efficient techniques for comprehensive protection from memory error exploits. In *USENIX Security Symposium*, 2005.

[13] Sandeep Bhatkar and R. Sekar. Data Space Randomization. In *Intl. Conf. on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[14] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *ACM Symp. on Information, Computer and Communications Security*, 2011.

[15] Peter Boonstoppel, Cristian Cadar, and Dawson R. Engler. RWset: Attacking path explosion in constraint-based test generation. 2008.

[16] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT – a formal system for testing and debugging programs by symbolic execution. 1975.

[17] Prashanth P. Bungale and Chi-Keung Luk. PinOS: a programmable framework for whole-system dynamic instrumentation. 2007.

[18] Shakeel Butt, Vinod Ganapathy, Michael M. Swift, and Chih-Cheng Chang. Protecting commodity operating system kernels from vulnerable device drivers. In *Annual Computer Security Applications Conf.*, 2009.

[19] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Symp. on Operating Systems Design and Implementation*, 2008.

[20] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *ACM Conf. on Computer and Communication Security*, 2006.

[21] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium*, pages 161–176, Washington, D.C., August 2015. USENIX Association.

[22] Nicholas Carlini and David Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, 2014.

[23] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Symp. on Operating Systems Design and Implementation*, 2006.

[24] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *ACM Symp. on Operating Systems Principles*, 2009.

[25] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *ACM Symp. on Operating Systems Principles*, 2009.

[26] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *ACM Conf. on Computer and Communication Security*, 2010.

[27] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M Frans Kaashoek, and Nickolai Zeldovich. Using crash hoare logic for certifying the fscq file system. In *25th Symposium on Operating Systems Principles (SOSP 15)*, pages 18–37. ACM, 2015.

[28] V. Chipounov and G. Candea. Enabling sophisticated analyses of x86 binaries with RevGen. In *Intl. Conf. on Dependable Systems and Networks (DSN)*, 2011.

[29] Vitaly Chipounov and George Candea. Reverse engineering of binary device drivers with RevNIC. In *ACM EuroSys European Conf. on Computer Systems (EUROSYS)*, 2010.

[30] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir, and George Candea. Selective symbolic execution. In *Workshop on Hot Topics in Dependable Systems (HOTDEP)*, 2009.

[31] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2011.

[32] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 30(1), 2012. Special issue: Best papers of ASPLOS.

[33] Andy Chou, Jun-Feng Yang, Bejamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. 2001.

[34] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: A software testing service. In *Workshop on Large Scale Distributed Systems and Middleware (LADIS)*, 2009.

[35] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. 2004.

[36] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointguardTM: protecting pointers from buffer overflow vulnerabilities. In *USENIX Security Symposium*, 2003.

[37] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, 1998.

[38] Code-pointer integrity. http://dslab.epfl.ch/proj/cpi/.

[39] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *ACM Conf. on Programming Language Design and Implementation*, 2002.

[40] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 191–206. ACM, 2015.

[41] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *USENIX Security Symposium*, 2014.

[42] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. 2008.

[43] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2008.

[44] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. SAFECode: enforcing alias analysis for weakly typed languages. *SIGPLAN Notices*, 41(6):144–157, June 2006.

[45] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[46] Artem Dinaburg and Andrew Ruef. Mcsema: Static translation of x86 instructions to llvm. In *ReCon 2014 Conference, Montreal, Canada*, 2014.

[47] George W. Dunlap, Dominic Lucchetti, Peter M. Chen, and Michael Fetterman. Execution replay on multiprocessor virtual machines. 2008.

[48] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. The matter of heartbleed. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 475–488. ACM, 2014.

[49] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. 2003.

[50] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Symp. on Operating Systems Design and Implementation*, 2006.

[51] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: Software guards for system address spaces. In *Symp. on Operating Systems Design and Implementation*, 2006.

[52] Isaac Evans, Sam Fingeret, Julián González, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. Missing the point(er): On the effectiveness of code pointer integrity. In *IEEE Symp. on Security and Privacy*, 2015.

[53] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *ACM Conf. on Programming Language Design and Implementation*, 2002.

[54] Malay K. Ganai and Aarti Gupta. Tunneling and slicing: towards scalable BMC. 2008.

[55] Archana Ganapathi, Viji Ganapathi, and David Patterson. Windows XP kernel crash analysis. 2006.

[56] Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *Intl. Conf. on Computer Aided Verification (CAV)*, 2007.

[57] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *ACM Conf. on Programming Language Design and Implementation*, 2005.

[58] Patrice Godefroid. Compositional dynamic test generation. In *Symp. on Principles of Programming Languages*, 2007.

[59] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *Network and Distributed System Security Symp. (NDSS)*, 2008.

[60] Patrice Godefroid and Daniel Luchaup. Automatic partial loop summarization in dynamic test generation. In *Intl. Symp. on Software Testing and Analysis (ISSTA)*, 2011.

[61] Enes Göktaş, Elias Athanasopoulosy, Herbert Bos, and Georgios Portokalidis. Out of control: Overcoming control-flow integrity. In *IEEE Symp. on Security and Privacy*, 2014.

[62] Trevor Hansen, Peter Schachte, and Harald Sondergaard. State joining and splitting for the symbolic execution of binaries. In *Intl. Conf. on Runtime Verification (RV)*, 2009.

[63] Niranjan Hasabnis, Ashish Misra, and R. Sekar. Light-weight bounds checking. In *IEEE/ACM Symp. on Code Generation and Optimization*, 2012.

[64] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. Ironclad apps: End-to-end security via automated full-system verification. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 165–181, Broomfield, CO, October 2014. USENIX Association.

[65] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. Fault isolation for device drivers. In *Intl. Conf. on Dependable Systems and Networks (DSN)*, 2009.

[66] Greg Hoglund and James Butler. *Rootkits: subverting the Windows Kernel*. Campus Press, 2006.

[67] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided automated software diversity. In *IEEE/ACM Symp. on Code Generation and Optimization*, 2013.

[68] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *IEEE Symp. on Security and Privacy*, 2013.

[69] Intel Architecture Instruction Set Extensions Programming Reference. http://download-software.intel.com/sites/default/files/319433-015.pdf, 2013.

[70] Intel. Introduction to Intel memory protection extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions, July 2013.

[71] Franjo Ivancic, Gogul Balakrishnan, Aarti Gupta, Sriram Sankaranarayanan, Naoto Maeda, Hiroki Tokuoka, Takashi Imoto, and Yoshiaki Miyazaki. DC2: A framework for scalable, scope-bounded software verification. In *ACM Intl. Conf. on Automated Software Engineering (ASE)*, 2011.

[72] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, Ilya Shlyakhter, and Pranav Ashar. F-soft: Software verification platform. In *Intl. Conf. on Computer Aided Verification (CAV)*, 2005.

[73] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conf.*, 2002.

[74] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. Tolerating hardware device failures in software. In *ACM Symp. on Operating Systems Principles*, 2009.

[75] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity softwar. In *Annual Computer Security Applications Conf.*, 2006.

[76] Sunghun Kim and E James Whitehead Jr. How long did it take to fix bugs? In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 173–174. ACM, 2006.

[77] James C. King. A new approach to program testing. 1975.

[78] James C. King. A new approach to program testing. 1975.

[79] Samuel T. King, George W. Dunlap, and Peter M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conf. (USENIX)*, 2005.

[80] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *ACM Symp. on Operating Systems Principles*, 2009.

[81] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, and Dawn Song. Poster: Getting the point (er): On the feasibility of attacks on code-pointer integrity. In *36th IEEE Symposium on Security and Privacy*, 2015.

[82] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-Pointer Integrity. In *Symp. on Operating Systems Design and Implementation*, 2014.

[83] Shuvendu K. Lahiri and Shaz Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *Symp. on Principles of Programming Languages*, 2008.

[84] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *IEEE/ACM Symp. on Code Generation and Optimization*, 2004.

[85] Chris Lattner and Vikram Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *ACM Conf. on Programming Language Design and Implementation*, 2005.

[86] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *ACM Conf. on Programming Language Design and Implementation*, 2007.

[87] K. Rustan M. Leino and Philipp Rümmer. A polymorphic intermediate verification language: Design and logical encoding. 2010.

[88] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, July 1993.

[89] Jinku Li, Zhi Wang, Tyler K. Bletsch, Deepa Srinivasan, Michael C. Grace, and Xuxian Jiang. Comprehensive and efficient protection of kernel control data. *IEEE Transactions on Information Forensics and Security*, 6(4):1404–1417, December 2011.

[90] The LLVM compiler infrastructure. http://llvm.org/.

[91] Ali Jose Mashtizadeh, Andrea Bittau, David Mazieres, and Dan Boneh. Cryptographically enforced control flow integrity. http://arxiv.org/abs/1408.1451, August 2014.

[92] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. 2005.

[93] Stephen McCamant and Greg Morrisett. Evaluating sfi for a cisc architecture. In *USENIX Security Symposium*, 2006.

[94] Kenneth L. McMillan. Lazy annotation for program testing and verification. In *Intl. Conf. on Computer Aided Verification (CAV)*, 2010.

[95] Manoel Mendonca and Nuno Neves. Fuzzing wi-fi drivers to locate security vulnerabilities. In *Dependable Computing Conference, 2008. EDCC 2008. Seventh European*, pages 110–119. IEEE, 2008.

[96] F. Mérillon, L. Réveillère, C. Consel, R. Marlet, and G. Muller. Devil: An IDL for hardware programming. In *Symp. on Operating Systems Design and Implementation*, 2000.

[97] Microsoft security advisory #944653: Vulnerability in Macrovision driver. http://www.microsoft.com/technet/security/advisory/944653.mspx.

[98] Microsoft. WHDC: Develop hardware for windows. http://www.microsoft.com/whdc, 2011.

[99] Brendan Murphy. Automating software failure reporting. *ACM Queue*, 2(8), 2004.

[100] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Intl. Symp. on Computer Architecture*, 2012.

[101] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. SoftBound: Highly Compatible and Complete Spatial Safety for C. In *ACM Conf. on Programming Language Design and Implementation*, 2009.

[102] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. CETS: Compiler Enforced Temporal Safety for C. In *Intl. Symp. on Memory Management*, 2010.

[103] G.C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Trans. on Programming Languages and Systems*, 27(3):477–526, 2005.

[104] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 11(58):http://phrack.com/issues.html?issue=67&id=8, November 2007.

[105] Ben Niu and Gang Tan. Monitor integrity protection with space efficiency and separate compilation. In *ACM Conf. on Computer and Communication Security*, 2013.

[106] Ben Niu and Gang Tan. Modular control-flow integrity. In *ACM Conf. on Programming Language Design and Implementation*, 2014.

[107] Vince Orgovan and Mike Tricker. An introduction to driver quality. Microsoft Windows Hardware Engineering Conf., 2003.

[108] Phoronix. Phoronix test suite. http://www.phoronix-test-suite.com/.

[109] The Associated Press. General electric acknowledges northeastern blackout bug. http://www.securityfocus.com/news/8032, 2004.

[110] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. Symdrive: Testing drivers without devices. In *Symp. on Operating Systems Design and Implementation*, 2012.

[111] Clang documentation: Safestack. http://clang.llvm.org/docs/SafeStack.html.

[112] 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 *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 745–762. IEEE, 2015.

[113] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *IEEE Symp. on Security and Privacy*, 2010.

[114] Margo Seltzer, Yasuhiro Endo, Christopher Small, and Keith Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Symp. on Operating Systems Design and Implementation*, 1996.

[115] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conf.*, 2012.

[116] O. Sery, G. Fedyukovich, and N. Sharygina. Interpolation-based function summaries in bounded model checking. In *HVC*, 2011.

[117] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *ACM Conf. on Computer and Communication Security*, 2007.

[118] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *IEEE Symp. on Security and Privacy*, pages 574–588, 2013.

[119] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: a lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conf. (USENIX)*, 2004.

[120] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering device drivers. *ACM Transactions on Computer Systems (TOCS)*, 24(4), 2006.

[121] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems (TOCS)*, 23(1), 2005.

[122] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. *IEEE Symp. on Security and Privacy*, 2013.

[123] The PaX team. Address space layout randomization. http://pax.grsecurity.net/docs/aslr.txt.

[124] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*, 2014.

[125] Arjan van de Ven. ExecShield. http://www.redhat.com/f/pdf/rhel/WHP0006US_Execshield.pdf, 2004.

[126] Jonas Wagner, Volodymyr Kuznetsov, and George Candea. -OVERIFY: Optimizing programs for fast verification. In *Workshop on Hot Topics in Operating Systems (HOTOS)*, 2013.

[127] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *ACM Symp. on Operating Systems Principles*, 1993.

[128] David Wheeler. SLOCCount. http://www.dwheeler.com/sloccount/, 2010.

[129] John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: Runtime intrusion prevention evaluator. In *Annual Computer Security Applications Conf.*, 2011.

[130] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *Symp. on Principles of Programming Languages*, 2005.

[131] Bin Xin, William N. Sumner, and Xiangyu Zhang. Efficient program execution indexing. In *ACM Conf. on Programming Language Design and Implementation*, 2008.

[132] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE Symp. on Security and Privacy*, 2009.

[133] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *USENIX Security Symposium*, 2013.

[134] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *ACM Conf. on Computer and Communication Security*, 2011.

[135] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. Practical Control Flow Integrity & Randomization for Binary Executables. In *IEEE Symp. on Security and Privacy*, 2013.

[136] J. Zhang. A path-based approach to the detection of infinite looping. 2001.

[137] Mingwei Zhang and R Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium*, 2013.

[138] Jianzhou Zhao, Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *ACM SIGPLAN Notices*, volume 47, pages 427–440. ACM, 2012.

[139] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. SafeDrive: safe and recoverable extensions using language-based techniques. In *Symp. on Operating Systems Design and Implementation*, 2006.