Compiler Optimizations with Retrofitting Transformations: Is there a Semantic Mismatch?

Jay P Lim
Rutgers University, USA
jpl169@scarletmail.rutgers.edu

Vinod Ganapathy
Indian Institute of Science, India
vg@iisc.ac.in

Santosh Nagarakatte
Rutgers University, USA
santosh.nagarakatte@cs.rutgers.edu

ABSTRACT

A retrofitting transformation modifies an input program by adding instrumentation to monitor security properties at runtime. These tools often transform the input program in complex ways. Compiler optimizations can erroneously remove the instrumentation added by a retrofitting transformation in the presence of semantic mismatches between the assumptions of retrofitting transformations and compiler optimizations. This paper proposes a strategy to ascertain that every event of interest that is checked in the retrofitted program is also checked after optimizations. Our initial experiments have identified bugs both in previously proposed retrofitting transformations and our implementations of retrofitting transformations.

1 INTRODUCTION

There is a vast body of prior work on monitoring program execution at runtime for enforcing various properties related to security, correctness, reliability, debugging, and many others [4, 5, 16, 17, 26, 33–38, 42, 49]. Typically, such monitoring is performed via retrofitting transformations, in which a given program is modified by adding instrumentation to perform checks, and propagate and store metadata information at runtime. For example, a transformation to detect buffer overflows or other memory-safety errors would instrument a program to propagate information about pointer bounds with each pointer operation and check them on pointer dereferences [5, 13, 16, 18, 36, 37, 46]. Similarly, a control-flow integrity enforcement mechanism would add instrumentation to propagate the set of allowed control flow transfers and check the target of every indirect call and jump instruction [4, 40].

Over the past several years, researchers and practitioners have developed numerous such retrofitting transformations. They differ widely in their design, ranging from transformations that are applied directly at the source-code level [13, 16, 23], to those that modify the compiler to add instrumentation [9, 10, 18, 26, 30, 37, 42, 46], and to those that add instrumentation via binary rewriting [14, 43, 44]. Regardless of the specific design used, retrofitting transformations are typically complex to implement. This is because the transformations are usually designed at an abstract level, but any real-world implementation must deal with the complexities of modern languages, runtime, and the hardware ABI.

Far from being research prototypes, such tools are now beginning to see wide-spread use. Sanitizers such as AddressSanitizer [42] for checking memory safety errors and CFI checkers [3, 45] are widely used compiler-based tools. Once a program has been instrumented with a retrofitting transformation, it is optimized with a suite of existing optimizations to reduce performance overheads following the "optimize-instrument-optimize" methodology [26, 32, 37, 38, 42, 46, 50]. The use of existing optimizations reduces the amount of new code added to optimize retrofitted programs while also minimizing performance overheads.

Apart from removing redundant checks, optimizations can remove a necessary check (from a retrofitting transformation perspective) when they optimize aggressively in the presence of undefined behavior [47]. Further, optimizations can also erroneously remove the inserted checks when implicit assumptions of the retrofitting transformation are not explicitly specified through the dataflow in the program (as these checks seem redundant from the optimization's perspective). For example, we discovered a bug in our SoftBoundCETS transformation in the presence of compiler optimizations due to a mismatch in the assumptions [1]. As a result, SoftBoundCETS propagates invalid bounds metadata in the shadow stack when the LLVM compiler optimizes function arguments after instrumentation [1]. Further, any semantic mismatch between correctness properties and the introduced checks can result in removal of checks [19]. One way to address this problem is to avoid compiler optimizations after retrofitting transformations. However, it can result in significant performance overheads. *Hence, this paper tackles an alternative research question: is the instrumentation added to the program by a retrofitting transformation still preserved after compiler optimizations?*

This paper proposes an approach to detect whether the added checks have been erroneously removed by optimizations due to mismatches in the assumptions. Our approach relies on the observation that any event of interest that is checked in the retrofitted program must also be checked in the resulting retrofitted program after optimizations. However, this task is challenging for the following reasons: (1) the checks can be safely optimized away when they are redundant, (2) the added instrumentation can be moved around with optimization, and (3) many small functions can be completely inlined (*e.g.*, with link time optimizations (LTO)).

To identify erroneously removed checks, we propose encoding the reachability of the event of interest as constraints in both the retrofitted program (P_{retro}) and optimized version of the retrofitted program (P^{opt}_{retro}). When the check is successful in the retrofitted program, the event of interest will be reachable and vice versa. We identify path conditions that makes the event of interest being

PLAS'17, October 30, 2017, Dallas, TX, USA

ACM ISBN 978-1-4503-5099-0/17/10…$15.00
https://doi.org/10.1145/3139337.3139343

reachable. Let's say, E_{retro} and E^{opt}_{retro} represents the constraints for event E to be reachable in the retrofitted program and the optimized version of the retrofitted program, respectively. Then, we check the validity of the formula $E_{retro} \implies E^{opt}_{retro}$ and $E^{opt}_{retro} \implies E_{retro}$. Our prototype addresses the challenges of encoding reachability with path conditions and checking the validity of these checks (see Section 2 for details). Our approach can be broadly viewed as a tailored application of translation validation, where the transformation applied to a single program is checked before and after optimizations.

We have developed an initial prototype to determine whether checks in retrofitted LLVM-IR programs have been erroneously optimized. We have tested the prototype with Olden benchmarks retrofitted with SoftBoundCETS [37, 38] and AddressSanitizer transformations. Our prototype detects erroneous removal of checks in a custom integer overflow checker with undefined behavior. It has also identified bugs when programs retrofitted with SoftBoundCETS transformation are optimized.

2 DETECTING ERRONEOUSLY REMOVED CHECKS

Programmers reasoning about a retrofitting transformation want to identify the added instrumentation. They may also want to ensure that the inserted instrumentation is not erroneously removed by the tool chain. Typically, retrofitting transformations are performed on a program after it has been optimized with a suite of existing optimizations. Subsequently, the program transformed with the retrofitted transformation is optimized again. This "*optimize-instrument-optimize*" methodology is widely used by many transformations [37, 42, 45, 46], which minimizes the amount of code added to perform retrofitting transformations.

The compiler or other parts of the tool-chain can remove checks when it infers them to be redundant or due to compiler bugs [2, 47]. This is especially true when parts of the check may not conform to the strict language standard (*i.e.* they may have undefined behavior according to the considered language standard) [29, 47]. However, the retrofitting transformation may consider the check to be essential. The problem is challenging because the developers of retrofitting transformations expect the tool-chain to remove redundant checks. Further, an optimized version of the retrofitted program can have little syntactic similarity to the retrofitted program especially with inlining and link time optimizations. Deploying completely verified tool-chains and verifying retrofitting transformations can address this problem. However, completely verified tool-chains are unavailable for mainstream systems. Hence, we propose a new approach to detect whether the tool-chain has erroneously removed the checks.

2.1 High-level Sketch

To identify erroneously removed checks, our approach is to identify events of interest, which varies with each retrofitting transformation, in the program that are protected by checks. We have to match the event of interest in the retrofitted program and its optimized version, which is a hard problem. To address this problem, we add a custom pass in the compiler that adds compile time metadata to the event of interest, which is maintained with optimizations.

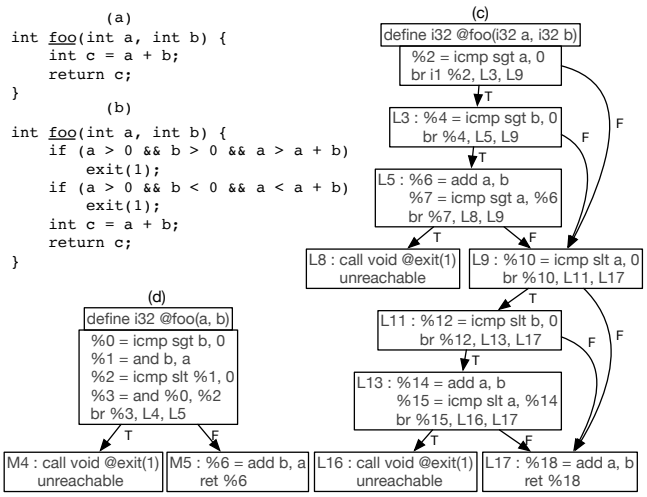

Figure 1: (a) A simple C program. (b) Naive integer overflow checks added to the C program in (a). (c) Simplified LLVM IR of (b). (d) Simplified LLVM IR of (c) after optimization (with -O1). LLVM optimized the integer overflow checks because $a > a + b$ evaluates to true when $a > 0$ and $b > 0$ and signed integer overflows are undefined behavior in C.

Using this compile-time metadata, we match these events in both retrofitted program (P_{retro}) and the optimized version of the retrofitted program (P^{opt}_{retro}). Once we identify the events of interest, we identify constraints that ensure the reachability of the event of interest in both the retrofitted program (P_{retro}) and the optimized version of the retrofitted program (P^{opt}_{retro}).

Conceptually, we encode all instructions from the beginning of the function till the event of interest and necessary path conditions as constraints. Let E_{retro} and E^{opt}_{retro} represent the constraints for event E to be reachable in P_{retro} and P^{opt}_{retro}, respectively. Subsequently we check if P_{retro} can reach an event of interest E, then event E should also be reachable in P^{opt}_{retro}; if P_{retro} cannot reach and event of interest E, then E should not be reachable even in P^{opt}_{retro}. Essentially, we check the validity of the following formula:

$$(E_{retro} \implies E^{opt}_{retro}) \wedge (\neg\, E_{retro} \implies \neg\, E^{opt}_{retro})$$

The contrapositive of $\neg E_{retro} \implies \neg\, E^{opt}_{retro}$ (*i.e.*, $E^{opt}_{retro} \implies E_{retro}$) is easier to compute and amenable for incremental generation of constraints. Hence, we check the validity of the formula:

$$(E_{retro} \implies E^{opt}_{retro}) \wedge (E^{opt}_{retro} \implies E_{retro})$$

Illustration. We illustrate our approach using a naive integer overflow checker. Figure 1 shows an integer overflow checker for C programs (naive but illustrates our point), a retrofitting transformation. Signed integer overflows are undefined behavior in C. Hence, the retrofitting transformation adds checks that aborts the program when an integer overflow occurs. The resultant LLVM IR for the retrofitted program is shown in Figure 1(c). The compiler evaluates the condition $a > a + b$ to true assuming that the program is well-defined (because signed integer overflow is undefined behavior)

and the resultant program (*i.e.* P^{opt}_{retro}) is shown in Figure 1(d). Here, the check in P^{opt}_{retro} can be considered to be erroneously removed.

The event of interest in P_{retro} is in block labeled L17 in P_{retro} (see Figure 1(c)) and in block labeled M5 in P^{opt}_{retro} (see Figure 1(d)). We generate reachability condition for this event in both P_{retro} and P^{opt}_{retro} and check the validity of the formula described above. Next, we illustrate the process of generating constraints to encode the reachability of the event.

2.2 Encoding Paths as Constraints

Our approach is based on the following observation: *when an optimization erroneously modifies or removes a check introduced by a retrofitting transformation, we will be able to observe that an execution may reach the event of interest in P_{retro} but aborts in P^{opt}_{retro}, and vice versa.* To detect that an event is reachable, we generate all distinct static paths that can reach the event in the program. We restrict ourselves to intra-procedural paths. We generate all possible paths in both P_{retro} and P^{opt}_{retro} that makes the event reachable. Next step is to generate constraints to generate the static path and relate the paths in P_{retro} and P^{opt}_{retro}.

First, we generate constraints for the static path to manifest. It consists of: (1) ϕ_{bc} — constraints to encode the branch conditions taken in the path, and (2) ϕ_{inst} — constraints that encode the sequence of non-branch instructions executed in the path. The constraint ϕ_{bc} is a conjunction of the constraints for each branch condition in the path. For example, there is only one path in P^{opt}_{retro} that reaches the event of interest in Figure 1(d). Hence, ϕ_{bc} is `%3 == false`. Similarly, the constraint ϕ_{inst} is a conjunction of constraints of each non-branch instruction in the path. Finally, the constraint for the path to manifest is $\phi_{bc} \wedge \phi_{inst}$.

Second, we generate a constraint that relates the paths in P_{retro} and P^{opt}_{retro}. Let's say $P0_{retro}$, $P1_{retro}$ ... PI_{retro} represent the constraints corresponding to the static paths in the retrofitted program (P_{retro}) that reaches the event of interest. Similarly, lets consider $P0^{opt}_{retro}$, $P1^{opt}_{retro}$ ... PJ^{opt}_{retro} represent the constraints for the static paths in the optimized retrofitted program (P^{opt}_{retro}) that reaches the event of interest. Then, we generate the following constraint to encode that if an event of interest is reachable in P_{retro}, then it is reachable in P^{opt}_{retro}:

$$(P0_{retro} \vee P1_{retro} \vee \ldots \vee PI_{retro}) \implies (P0^{opt}_{retro} \vee P1^{opt}_{retro} \vee \ldots \vee PJ^{opt}_{retro}).$$

Similarly, we generate constraints to encode that if the event is not reachable in P_{retro}, then it is not reachable in P^{opt}_{retro}. We use the contrapositive of the above statement:

$$(P0^{opt}_{retro} \vee P1^{opt}_{retro} \vee \ldots \vee PJ^{opt}_{retro}) \implies (P0_{retro} \vee P1_{retro} \vee \ldots \vee PI_{retro}).$$

The final constraint that we generate is the conjunction of the above two conditions. Finally, we also generate constraints to relate the initial memory states, function arguments, and live registers of P_{retro} and P^{opt}_{retro}.

We repeat this process for every event of interest in the retrofitted program. Our approximations in relating memory states, encoding paths involved with loops and function calls can result in both missing errors and false errors. Our design choices try to minimize such false errors while being useful in detecting erroneously removed checks.

Incremental construction of queries. SMT solvers can quickly solve small queries. The size of the queries will increase with an increase in the number of branches and instructions considered. Hence, we incrementally construct the query and break the above formula checked for validity into smaller parts. Mathematically, $(p_1 \vee p_2) \implies q$ is equivalent to $(p_1 \implies q) \wedge (p_2 \implies q)$ and $p \implies (q_1 \vee q_2)$ is equivalent to $(p \implies q_1) \vee (p \implies q_2)$.

Therefore, one part of the validity check:

$$(P0_{retro} \vee P1_{retro} \vee \ldots \vee PI_{retro}) \implies (P0^{opt}_{retro} \vee P1^{opt}_{retro} \vee \ldots \vee PJ^{opt}_{retro})$$

can be simplified to

$$((P0_{retro} \implies P0^{opt}_{retro}) \vee \ldots \vee (P0_{retro} \implies PJ^{opt}_{retro})) \wedge \ldots \wedge ((PI_{retro} \implies P0^{opt}_{retro}) \vee \ldots \vee (PI_{retro} \implies PJ^{opt}_{retro})).$$

Abstractly, this equation states that for every path in P_{retro}, there must be a corresponding path in P^{opt}_{retro}. For example, in Figure 1, there are 9 distinct paths that can reach the event of interest in P^{opt}_{retro} and 1 path that reaches the event of interest in P_{retro}. Hence, one part of the validity check is equivalent to $(P0_{retro} \implies P0^{opt}_{retro}) \wedge (P1_{retro} \implies P0^{opt}_{retro}) \wedge \ldots \wedge (P8_{retro} \implies P0^{opt}_{retro})$. This simplification was instrumental in scaling our detector to large functions.

3 EVALUATION

This section describes the methodology used for generating retrofitted programs, optimized retrofitted program, and detecting erroneous removal of checks.

3.1 Prototype and Methodology

We have built a prototype tool to detect erroneously removed checks for retrofitted and optimized retrofitted programs expressed as LLVM IR programs. Although we evaluated our prototype with programs expressed as LLVM IR programs, it can also be used with x86-binaries. We modified retrofitting transformations to mark all events of interest (using compiler metadata) for this evaluation. We check the validity of the queries generated by our prototype using the Z3 [15] SMT solver.

We evaluated our prototype by using it to detect erroneously removed checks in Olden benchmarks retrofitted using the SoftBoundCETS [32] pass, AddressSanitizer [42] transformation, and our custom integer overflow checking transformation. We built a custom integer overflow checker to test the effectiveness of the tool when the checks introduced by the retrofitting transformation leverages some form of undefined behavior, which allows the LLVM optimizer to remove parts of the check. All these retrofitting transformations are LLVM-based transformations that work on the source code of P_{orig} and produce P_{retro} when the compiler is invoked with the corresponding flags. We configured the tools to use LLVM optimization level `O3` with link time optimizations to produce both P_{orig} and P_{retro}.

We used nine Olden benchmarks transformed with retrofitting transformations for this evaluation. To ensure that we do not experience timeouts while checking the validity of queries, we restricted the prototype to consider events that have at most 10 static paths and have approximately 200 LLVM IR instructions in either P_{retro} or P^{opt}_{retro}.

Address Sanitizer					
Benchmark	Functions	Total Events	Check Success	Check Failed	Time-Out
bh	38	135	121	9	5
bisort	7	20	18	1	1
em3d	9	27	21	4	2
health	13	41	37	3	1
mst	10	14	11	2	1
perimeter	6	28	27	1	0
power	13	56	48	2	6
treeadd	6	8	6	2	0
tsp	9	19	16	2	1

SoftBoundCETS					
Benchmark	Functions	Total Events	Check Success	Check Failed	Time-Out
bh	52	263	259	2	2
bisort	16	86	86	0	0
em3d	15	66	63	3	0
health	17	108	105	3	0
mst	16	88	85	1	2
perimeter	11	82	78	4	0
power	6	25	22	2	1
treeadd	8	43	43	0	0
tsp	13	84	84	0	0

Table 1: Table presents the data on the number of functions where the tool checked for erroneous removal of checks, total number of events of interest checked in the application, the number of events for which tool successfully validated that the check was not removed, the number of events where tool could not successfully validate that the check was not removed (*i.e.*, either a false positive or a true error), and the number of instances when solver experienced time outs with both SoftBoundCETS and AddressSanitizer.

3.2 Effectiveness in Detecting Erroneously Removed Checks

Table 1 reports the number of functions that were used as part of the evaluation, the number of events that were checked in those functions, the total number of events where our tool was successfully able to confirm that the check was not removed, total number of events where our tool could not confirm that the check was removed, and the number of timeouts experienced during our evaluation. Whenever our tool could not confirm that the check was removed, it could be because either the check was removed erroneously or it is a false positive. We have not completely examined all failed checks.

The false positives are typically due to the length of the query and approximation in the assumptions encoded with our preconditions. For example, SoftBound/CETS uses multiple levels of disjoint metadata. We need to ensure that any alloca slot is disjoint from any entry in the disjoint metadata space. Our implementation makes some approximation in encoding the metadata structure as constraints, which can result in false positives. Our goal was to determine if a programmer who is not aware of the internal details of the retrofitting transformation can identify whether checks are erroneously removed. Hence, we did not encode the details about the metadata layout in our constraints.

While we were building our prototype, we discovered a bug in the SoftBoundCETS implementation. The root cause of the bug was due to a semantic mismatch between the assumptions of the SoftBoundCETS transformation and actions taken by compiler optimizations. SoftBoundCETS transformation passes metadata for pointer arguments using a shadow stack. It uses the position of the argument in the function signature of the called function to retrieve metadata from the shadow stack. When the optimizer removes an argument (which changed the function signature), the SoftBoundCETS checks would get invalid metadata because the position of the pointer argument has changed. Subsequently, we have created a micro benchmark with a function that takes two function arguments. The first argument of the function is optimized away by the compiler, which our tool was able to detect. We have confirmed that the SoftBoundCETS bug is a valid bug [1], which is one of the root causes of false positive memory safety errors reported by SoftBoundCETS. Our prototype also detected all instances of erroneous removal of checks in the presence of optimizations with our integer overflow checker.

4 RELATED WORK

There is a large body of work on reverse-engineering, malware analysis, and binary analysis. We highlight the most related work.

Translation Validation. Translation validation checks the correctness of a compiler optimization for a given program rather than checking the correctness of the optimization for all programs [39, 41]. Translation validation is attractive because it is easier to check the transformation for a single instance rather than proving correctness. Our approach can be considered similar in spirit to translation validation because our technique determines whether checks have been erroneously removed in a single program. However, we address false positives in a generic translation validator by designing custom procedure for detecting such erroneously removed checks.

Semantic Differencing. Symdiff [27] proposes a language-agnostic tool for checking equivalence and semantic difference of imperative programs. The resultant programs generated from retrofitting transformations can be checked with the original program for equivalence and differences. To use Symdiff, one would have to enhance it with detailed semantics of retrofitting transformations. Otherwise, it would report any added instrumentation as a semantic difference. In contrast, this paper addresses a more directed problem of determining whether optimizations have erroneously removed instrumentation without requiring detailed information about retrofitting transformations.

Detecting Undefined Behavior. Compilers optimize assuming there is no undefined behavior. Checks are typically removed if they rely on any undefined behavior. Stack [47] detects the stability of programs with undefined behavior by interpreting the program under two language semantics and checking if they have diverging behavior. It may be possible to detect removal of checks with Stack. In contrast to Stack, our approach can detect any check that is eliminated through optimization not just in the presence of undefined behavior.

Binary Analysis Tools. Statically analyzing x86 binary code is a well-studied area with a variety of goals and applications (*e.g.* malware detection [11, 12]). Binary analysis tools can possibly be used

to reason about LLVM IR code. However, the precision of such techniques is a concern.

Binary differencing is related to our work and has important applications, such as in detecting security holes or generating software fingerprints [8]. Among the early tools in this area was BinDiff [20], which proposed techniques to identify similarity between two executables using graph isomorphism. BinDiff normalized binaries as control flow graphs (CFG) and explored similarity and differences between CFGs. BinHunt [21] and BinSlayer [7] extend BinDiff's algorithms with a combination of symbolic execution, theorem proving and bipartite matching. Rendezvous [24] uses statistical model along with CFGs to enable searching for binary code. BitShred [22] uses feature hashing to cluster similar binary components.

Dynamic binary Analysis has successfully been applied for a number of reverse-engineering tasks, such as for binary component reuse (*e.g.* [25, 51]) and forensics applications (*e.g.* [28]). It is also possible to use dynamic analysis to reason about concrete executions and extract semantically-similar code. In the past, even simple clustering of runtime execution traces has shown promise for tasks such as malware classification [6]. Dynamic similarity detection techniques can leverage control-flow matching techniques [31] or more sophisticated forms of execution indexing [48]. In contrast to dynamic approaches, our tool can detect semantically-similar transformations of instrumentation code statically without the need for inputs to the program.

5 CONCLUSION

Retrofitting transformation change programs in complex ways. Compiler optimizations can remove the added instrumentation as a result of implicit assumptions, semantic mismatches about the necessity of the instrumentation, and/or undefined behavior in the code. Our prototype offers programmers the ability to understand precisely whether the added instrumentation persists after optimizations. We have proposed a novel approach to encode the property that added checks have not been erroneously removed as constraints. Our initial results show that these approaches are useful in detecting bugs that arise in retrofitting programs with compiler optimizations.

ACKNOWLEDGMENTS

This paper is based on work supported in part by NSF CAREER Award CCF–1453086, a sub-contract of NSF Award CNS–1116682, a NSF Award CNS–1441724, and NSF Award CNS–1408803.

REFERENCES

[1] 2016. SoftBoundCETS shadow stack metadata propagation is wrong when llvm optimizations remove arguments. (2016). https://github.com/santoshn/softboundcets-34/issues/8.

[2] 2017. 2017/04/17 0 apple clang elides bounds check. (2017). https://github.com/sandstorm-io/capnproto/blob/master/security-advisories/2017-04-17-0-apple-clang-elides-bounds-check.md.

[3] 2017. Windows Dev Center, Control Flow Guard. (2017). https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-Flow Integrity. In *ACM Conference on Computer and Communications Security (CCS)*.

[5] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *USENIX Security Symposium*.

[6] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. 2009. Scalable, Behavior-Based Malware Clustering. In *Network and Distributed System Security Symposium (NDSS)*.

[7] Martial Bourquin, Andy King, and Edward Robbins. 2013. BinSlayer: Accurate Comparison of Binary Executables. In *Program Protection and Reverse Engineering Workshop (PPREW)*.

[8] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. 2007. Towards Automatic Discovery of Deviations in Binary Implementations with Applications to Error Detection and Fingerprint Generation. In *USENIX Security Symposium*.

[9] Scott A. Carr and Mathias Payer. 2017. DataShield: Configurable Data Confidentiality and Integrity. In *ACM Sympsium on Information, Computer and Communications Security (ASIACCS)*.

[10] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing software by enforcing data-flow integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[11] Mihai Christodorescu and Somesh Jha. 2003. Static Analysis of Executables to Detect Malicious Patterns. In *USENIX Security Symposium*.

[12] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. 2005. Semantics-aware malware detection. In *IEEE Symposium on Security and Privacy*.

[13] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. 2003. CCured in the Real World. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[14] Lucas Davi, Ra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan NÃijrnberger, and Ahmad reza Sadeghi. 2012. MoCFI: A framework to mitigate control-flow attacks on smartphones. In *Network and Distributed System Security Symposium (NDSS)*.

[15] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*.

[16] Christian DeLozier, Richard Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo M. K. Martin, and Steve Zdancewic. 2013. Ironclad C++: A library-augmented type-safe subset of C++. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*.

[17] Dinakar Dhurjati and Vikram Adve. 2006. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In *International Conference on Software Engineering (ICSE)*. 162–171.

[18] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[19] Vijay D'Silva, Mathias Payer, and Dawn Song. 2015. The correctness-security gap in compiler optmization. In *IEEE Symposium on Security and Privacy Workshops*.

[20] Halvar Flake. 2004. Structural comparison of executable objects. In *SIG SIDAR Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[21] Debin Gao, Michael K. Reiter, and Dawn Song. 2008. BinHunt: Automatically Finding Semantic Differences in Binary Programs. In *International Conference on Information and Communications Security (ICICS)*.

[22] Jiyong Jang, David Brumley, and Shobha Venkataraman. 2011. BitShred: Feature Hasing Malware for Scalable Triage and Semantic Analysis. In *ACM Conference on Computer and Communications Security (CCS)*.

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

[24] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. 2013. Rendezvous: A Search Engine for Binary Code. In *Working Conference on Mining Software Repositories (MSR)*.

[25] Dohyeong Kim, William N. Sumner, Xiangyu Zhang, Dongyan Xu, and Hira Agrawal. 2014. Reuse-oriented Reverse Engineering of Functional Components from x86 binaries. In *International Conference on Software Engineering (ICSE)*.

[26] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[27] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-agnostic Semantic Diff Tool for Imperative Programs. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12)*.

[28] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Network and Distributed System Security Symposium (NDSS)*.

[29] Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. 2016. Into the Depths of C: Elaborating the De Facto Standards. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[30] Daniele Midi, Mathias Payer, and Elisa Bertino. 2017. Memory Safety for Embedded Devices with nesCheck. In *ACM Sympsium on Information, Computer and Communications Security (ASIACCS)*.

[31] Vijay Nagarajan, Rajiv Gupta, Xiangyu Zhang, Matias Madou, Bjorn de Sutter, and Koen de Bosschere. 2007. Matching Control Flow of Program Versions. In *International Conference on Software Maintenance (ICSM)*.

[32] Santosh Nagarakatte. 2012. *Practical Low-Overhead Enforcement of Memory Safety for C Programs*. Ph.D. Dissertation. University of Pennsylvania.

[33] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*.

[34] Santosh Nagarakatte, Milo M K Martin, and Steve Zdancewic. 2013. Hardware-Enforced Comprehensive Memory Safety. IEEE MICRO 33, 3 (May/June 2013).

[35] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2014. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14*. 175.

[36] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything you want to know about pointer-based checking. In *Proceedings of SNAPL: The Inaugural Summit On Advances in Programming Languages*.

[37] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the SIGPLAN 2009 Conference on Programming Language Design and Implementation*.

[38] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*.

[39] George Necula. 2000. Translation validation for an optimizing compiler. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[40] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[41] Hanan Samet. 1978. Proving the correctness of heuristically optimized code. *Communications of the ACM (CACM)* (1978).

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

[43] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Security Symposium*.

[44] Julian Seward and Nicholas Nethercote. 2007. Valgrind: a framework for heavy-weight dynamic binary instrumentation.. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

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

[46] Jonas Wagner, Volodymyr Kuznetsov, George Candea, and Johannes Kinder. 2015. High system-code security with low overhead. In *IEEE Symposium on Security and Privacy*.

[47] Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards optimization-safe systems: Analyzing the impact of undefined behavior.. In *ACM Symposium on Opererating Systems Principles (SOSP)*.

[48] Bin Xin, William N. Sumner, and Xiangyu Zhang. 2008. Efficient Program Execution Indexing. In *ACM Conference on Programming Language Design and Implementation (PLDI)*.

[49] Wei Xu, Daniel C. DuVarney, and R. Sekar. 2004. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*.

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

[51] Junyuan Zeng, Yangchun Fu, Kenneth A. Miller, Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2013. Obfuscation-resilient binary code reuse through trace-oriented programming. In *ACM Conference on Computer and Communications Security (CCS)*.