Beyond Binary Program Transformation

Bruno Dutertre
SRI International
333 Ravenswood Ave
Menlo Park, CA
bruno@csl.sri.com

Ashish Gehani
SRI International
333 Ravenswood Ave
Menlo Park, CA
gehani@csl.sri.com

Hassen Saidi
SRI International
333 Ravenswood Ave
Menlo Park, CA
saidi@csl.sri.com

Martin Schäf
SRI International
333 Ravenswood Ave
Menlo Park, CA
schaef@csl.sri.com

Ashish Tiwari
SRI International
333 Ravenswood Ave
Menlo Park, CA
tiwari@csl.sri.com

ABSTRACT

We describe a number of program transformation and analysis tools developed at SRI to tackle bloatware from three angles: slicing binaries to exclude unnecessary components, transformation of different copies of the same binary to create diversity and reduce the potential impact of an attack, and verification-based super-optimization to prune unreachable code and harden vulnerable code.

At the core of these approaches is code transformation: we must add, remove, or change a binary without altering its desired behavior (but potentially removing some of its undesired behavior). The challenge is to ensure that these transformations are correct. We argue that a formal model of intermediate code representation (e.g., LLVM or JVM bytecode) is the right level of abstraction to address this challenge.

Keywords

De-compilation; Code Transformation; Rewriting

1. INTRODUCTION

The modern software stack is constantly increasing in complexity. Web, mobile, cloud, and distributed applications are often built as layers of frameworks and libraries to ensure coordination, portability, and scalability in an environment where software, platforms, and services are offered as increasingly complex and opaque layers of services. Indeed, in 2016, you can build software without writing an original piece of code. Just plug in to your favorite code library or package registry, and cut, paste, and compile. As a result, it is becoming harder to ensure the integrity of modern software.

A number of recent bugs and security vulnerabilities illustrates how bloatware and lack of visibility into software dependencies are making our computing infrastructure increasingly fragile.

Bringing the Internet to its knees. Earlier this year, one developer broke thousands of web projects and created a panic because of eleven lines of JavaScript code. Azer Koculu removed more than 250 of his modules from NPM—a popular package manager used by JavaScript projects—because of brand infringement claims from Kik. In this process, Koculu removed eleven lines of JavaScript code named `left-pad` that pads out the lefthand-side of strings with zeros or spaces. The removal of `left-pad` almost brought the Internet to its knees, as it was in the dependencies of Node, React, Babel, and other popular packages for web development. A closer look at the NPM ecosystem reveals an even worse dependency situation. A fresh install of the Babel package includes 41,000 files, and `jspm/npm`-based app templates now starts with 28,000+ files. This may be a case of poor package management, but it illustrates how hidden, unnecessary dependencies plague modern software stacks.

Breaking Web Encryption. FREAK (”Factoring RSA Export Keys”) exploits a weakness in the SSL/TLS protocols introduced decades earlier for compliance with U.S. cryptography export regulations. These limited exportable software to use only public key pairs with RSA moduli of 512 bits or less (so-called `RSA_EXPORT` keys). By the early 2010s, increases in computing power meant that 512bit RSA keys could be broken by anyone with access to relatively modest computing resources, at a cost of less than \$100 of cloud computing services. This meant that a man-in-the-middle attack, with only a modest amount of computation, could break the security of any website that allowed the use of 512-bit export-grade keys. This exploit was publicly revealed in 2015, but the underlying vulnerabilities had been present for many years, dating back to the 1990s.

Breaking Linux. Last year, researchers have discovered a potentially catastrophic flaw in one of the Internet’s core building blocks. This leaves hundreds or thousands of apps and hardware devices vulnerable to attacks that can take complete control over them. The vulnerability was introduced in 2008 in the GNU C Library, a collection of open source code that powers thousands of standalone applications and most distributions of Linux, including those distributed with routers and other types of hardware. Function `getaddrinfo` that performs domain-name lookups contains a buffer overflow bug that allows attackers to remotely execute

FEAST ’16 Vienna, Austria
© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

malicious code. The bug can be exploited when vulnerable devices or apps make queries to attacker-controlled domain names or domain name servers, or when they are exposed to man-in-the-middle attacks. All versions of `glibc` after 2.9 were vulnerable. The widely used secure shell, sudo, and curl utilities are all known to be vulnerable, and the list of other affected apps or code is almost too diverse and numerous to fully enumerate. This bug hit the core bedrock of any Linux distribution.

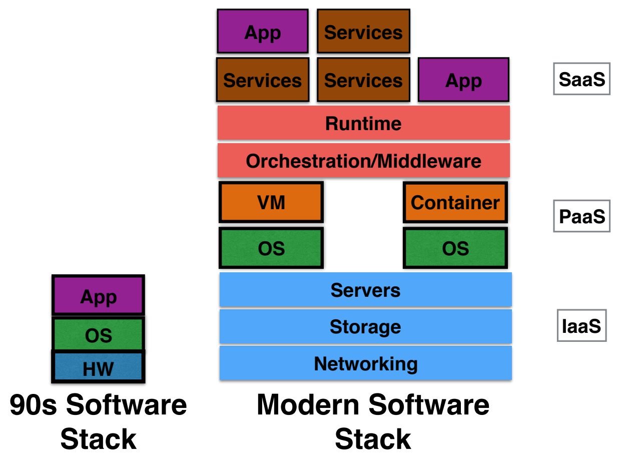

Figure 1: Modern Complex Software Stack

Protect Locally. What makes the scenarios above so dangerous is that the underlying bugs are found in commonly used libraries whose code can be nested deep inside binaries. In this situation, the user must rely on software vendors or developers to keep track of components on which their product depends and update the dependencies when vulnerabilities are found. With the ever increasing size of the technology stack inside a single application, this dependency on the integrity of subcomponents becomes a severe risk.

Even if each vendor of a library reacts immediately once a new security vulnerability is identified, it takes time to develop a fix and inform the users that they should upgrade. As the length of the dependency chain increases, it takes more and more time to fix all binaries. During this entire time, users are exposed.

To address this risk, we need means to analyze and transform binaries locally, on our own machine, without relying on vendor updates. In Section 2, we discuss three angles from which we try to mitigate this risk: slicing binaries to exclude components that are not required, transformation of different copies of the same binary to create diversity and reduce the potential impact of an attack, and verification-based super-optimization to prune unreachable code and harden vulnerable code.

At the core of these approaches is code transformation: we must add, remove, or change a binary without altering its desired behavior (but potentially removing some of its undesired behavior). The challenge is to ensure that these transformations are correct. For local binaries, we cannot rely on the programmer to check the result, and we usually do not have regression tests to reassure us that the transforms preserve the relevant behavior.

2. PROTECT LOCALLY

A straight forward approach to reduce the risk of unknown zero-day exploits in bloated binaries is to slim down the binary, exclude unnecessary functionality, and to do some basic re-writing to make the binary less susceptible to generic attacks.

In the following, we discuss how we achieve this goal using three different types of tools that we develop at SRI and how these approaches can benefit from a shared infrastructure and what quality assurance mechanisms can be used when developing these tools.

2.1 Code Winnowing

OCCAM [16] (Object Culling and Concretization for Assurance Maximization) is a software winnowing tool. Its goal is to reduce software bloat by automatically removing unused functionalities. OCCAM is based on LLVM 3.5. It performs dependency and control-flow analysis to detect and remove unused functions from software. It can also specialize existing software by employing partial evaluation (i.e., fix some arguments of a function call), which can lead to further removal of unused code. OCCAM can work across multiple LLVM modules by computing and summarizing cross-module dependencies, applying specialization where appropriate, and removing unreachable code.

2.2 Assured Program Transforms

As part of the DARPA CFAR program, in collaboration with the University of Virginia, we are developing formal verification techniques for proving that various program transforms and rewriting preserve program semantics [6]. Program rewriting introduces artificial code diversity to reduce vulnerabilities. For example, some program transforms may randomly relocate code snippets to address that we hope are not guessable by an adversary. These transforms are intended to make ROP attacks difficult if not impossible. In CFAR, an additional level of security is obtained by running several, diverse variants of the code run in parallel and cross-check each other. This provides a means to detect attacks (assuming diversity makes it impossible for the same attack to succeed on all variants).

In this project, SRI is investigating formal verification methods and tools for proving that code rewriting preserves functional correctness and for showing that some classes of attacks are prevented. We rely on SRI's formal method tools such as the PVS theorem prover, and the Yices SMT Solver. We have developed a state-machine model of the target hardware (namely, the x86_64 processor, and we are developing proof methods based on so-called translation validation. Our models are executable and can be validated via testing.

2.3 Verification-based Optimization

Optimization of binaries has two essential steps: identifying and eliminating parts of an application that are not needed (under some condition), and super-optimizing the remaining part of the application.

Conditional reachability can be checked using static verifiers at a higher precision than with the methods that are built into a compiler. A verifier can augment existing reachability checks while also considering path feasibility and thus is able to safely prune parts of an application that cannot be pruned using a compiler.

Synthesis techniques used in formal verification, can be used to identify optimal sequences of loop-free code. Synthesis-based super-optimization often outperforms existing approaches that rely on heuristics because they can guarantee that an optimal solution will be found.

In the following we describe tools developed by SRI that implement these two approaches and discuss how they make use of program transformations and the strategies we use to ensure the quality of these transformations.

Proving Unreachability. Knowing that code cannot be reached is an essential part of program optimization. Once we know that code will not be executed, we can drop it from an application. This does not only result in performance and space improvements but also eliminates potential attack vectors and places where malicious code could hide.

Software verification can be used to identify unreachable code at a much higher precision than existing complier infrastructures and can also check conditional unreachability, for example, under a given precondition. Such techniques have been used successfully to identify and eliminate code that is unstable under certain compiler-optimizations [22] and to identify unreachable code that may be vulnerable [17]. Further, these techniques can be used to identify and eliminate unnecessary error handling which can yield considerable performance improvements [4]. These tools perform a sound reachability analysis on an over-approximation of the reachable program states. Soundness guarantees that code that cannot be reached by this analysis is in fact unreachable.

At SRI we work on two static verification based tools to detect unreachable code: SeaHorn [11] which analyzes LLVM-IR and Bixie [17] which analyzes Java bytecode. Both tools use static verification techniques to identify code that is provably unreachable. The tools have identified unreachable code (and bugs related to that) in big open-source projects such as Tomcat.

Synthesizing Super-Optimized Code. Automated program synthesis refers to the automatic generation of programs from high-level specifications. Program synthesis has recently seen renewed interest and tremendous progress. Moreover, its effectiveness has been demonstrated on plethora of applications, including the Microsoft FlashFill feature in Microsoft Excel software [9]. We performed some early work on *component-based synthesis* [10], where the goal is to generate a program that uses only the given set of components. A particular application of the component-based synthesis paradigm was super-optimization. Here, the unoptimized code is seen as a "specification", and the goal is to synthesize code that is equivalent to this specification. The synthesis procedure assumes a bound on the length of the program, and uses components from a given library of components. We showed that the component-based synthesis problem can be reduced to an exists-forall constraint, which can be efficiently solved using an "counter-example guided inductive synthesis" approach [10]. The approach performs much better than the brute-force approach of enumerating all possible programs, which is typically used for performing super-optimization. Rather than enumerate all candidate programs, the counter-example guided approach navigates the search space of programs guided by learning from earlier failures. In fact, our algorithm for component-based synthesis has recently been used in the super-optimizer for LLVM IR called Souper [14]. We have also developed a stand-alone synthesis tool, called Synudic [8, 20], which is an implementation of an extended version of the original component-based synthesis procedure. The implementation uses the SMT solver, Yices [7], as its backend. We note here that Yices now supports limited reasoning on exists-forall formulas.

3. HIGHER-LEVEL TRANSFORMATIONS

All defense mechanisms that we discussed in Section 2 rely on program transformation of low-level code. Such transformations traditionally are composed of two parts: a decompilation step that lifts the code to a higher level representation by recovering information about structure and intent (such as high-level types) of the low-level code, and a re-writing step that applies the desired transformation to higher-level code before recompiling it to its original low-level form.

To ensure quality of program transformation we strictly separate these two steps. Assuring quality of a decompilation step is inherently different than assuring quality of a re-writing step: for the decompilation, we know that we do not want to change the behavior of the program. We can, for example test equivalence of a program and a version of it after de- and re-compilation, and we could use symbolic techniques to generate test inputs that compare the IR representation with the binary.

Once we have a reliable and testable decompilation step, re-writing code becomes much simpler and less error prone. High-level code such as LLVM-IR also comes with much better tools support for testing and debugging. However, testing the correctness of re-writing is more complicated since it may change program behavior and we cannot apply equivalence checking. In the following we discuss SRI's tools for decompilation and program re-writing together with the steps we take to ensure quality.

3.1 Decompilation

Decompilation is a program transformation by which a high-level source code for an executable program is discovered. Decompilation techniques were initially used to aid in the migration of programs from one platform to another. Since then, decompilation techniques have been used to aid in the recovery of lost source code, debugging of programs, locating of viruses, comprehending programs, recovery of high-level views of programs, and more. The ability to extract for instance C code from `x86` code allows the application of a large array of source code analysis tools to a binary. Indeed, there are more tools for source code analysis than binary analysis. Decompilation is considered primarily as a comprehension aid to find bugs, vulnerabilities, malicious code, and verification. But recent applications [15] extend to code search and mining repositories of software as well. Most current decompilers aim at program comprehension and do not tackles the problem of producing code that compiles and can be targeted to different platforms. We at SRI, have attempted to solve this problem by addressing the main deficiency of modern decompiler. Namely, the analysis of data sections to properly account for data types, and avoid memory segmentation errors when running the decompiled code.

The most advanced decompiler today is the Hex-Rays decompiler [1]. It generates readable C code from a variety of sources including x86 and ARM binaries. The decompiler performs the following tasks:

- It first relies on the disassembler to identify code and data in a binary.
- It identifies function boundaries, and associates with each assembly function, a corresponding C function.
- For each function, it identifies using stack pointer analysis the calling convention, the local variables, and arguments.
- For each function assembly code, it generate a single static assignment representation, and proceeds to simplify the code by eliminating registers and interpreting low level instructions as C statements using only local and global variables and function arguments.

Without performing further type analysis, it is possible to obtain a readable C code using these steps. Since C is pretty much close to assembly code, it is possible to interpret most variables and arguments as 32 and 64 bits integers for `x86` and `x86_64` respectively. However, the data section is often misinterpreted by Hex-Rays. The SRI OpensDec (Open Source Decompiler) project [2] implements a series of binary transformations that allows for the proper decompilation of code, and the proper interpretation of the data sections so that the decompiled code can be recompiled without risks of runtime errors.

Recently, another form of decompilation of binaries is gaining significant ground. The ability of transforming binary code into the intermediate representation (IR) or LLVM allows the application of a wide range of transformation and optimization passes. It also allows for the re-targeting of the code to multiple platforms. The translation is based on the same principles as with decompilers. That is, using function boundary, arguments, and local variables to translate every function into an LLVM IR function. A new promising approach consists of using Machine Translation [13] to build decompilation capabilities. It is possible to apply techniques for source to source transformation between different programming languages to assembly and LLVM IR. The decompilers built-in transformations are generated by machine learning.

3.2 Program Transformations

With the recent trend of performing program analysis directly on the LLVM bitcode or Java bytecode, this step is becoming more popular due to the easy availability of transformation toolkits such as Soot [21], Wala [3], Cil [18], and a variety of tools inside Clang and LLVM.

Analyzing intermediate code has many advantages. For example, in the case Java, the source code has over 50 different keywords and countless constructs (e.g., try-with-resources) that are hard to handle. The JVM bytecode still has way over one hundred different cases that need to be considered. The Jimple [21] intermediate language, on the other hand, that is used inside Soot has less than 20 cases that need to be considered. Further, Jimple already provides a 3-address representation. Similar simplifications can be found in other frameworks.

On these 3-address code programs, we take two approaches to ensure the correctness of transformations: unit fuzzing and manual verification. In many cases, unit fuzzing is a low cost approach to ensure correctness of a transformation. For example, in our JayHorn [12] and Bixie [17] tool, we employ a transformation that makes exceptional control-flow explicit (by introducing a global that tracks the last exception and checking if this global is set).

Verifying such a transformation is hard or even impractical since we actually do not eliminate all exceptional behavior (e.g. JVM errors may still occur). Formalizing such a transformation would be too complicated to be useful. Writing complex formalizations is an error prone process so the value of the proof will be questionable.

Instead, we systematically test our transformation to reduce risk. On a whole program level, this transformation is behavior preserving but not for individual units (since a previously exceptional return only updates the global and then returns a default value). To ensure that such a transformation works properly, we write a light-weight test driver that takes a method (or classes), wraps it in a catch-all block and returns a string representation of the method's return value or of the thrown exception. Then we employ a black-box fuzzer (here Randoop [19]) to generate input-output pairs. Now we can use these input output pairs to test our transformation.

For other transformations, it is possible, or even worth it to verify their correctness against a formal model of the language using an interactive proof assistant such as PVS. Again, operating on a simplified 3-address code version of the program is crucial. Getting a formal model for example for x86 assembler is very complicated and error prone. We can partially validate such a model by testing or simulation, but any proof of a transformation based on this model strongly relies on our confidence in the person that wrote it. In our experience, LLVM IR is much easier to formalize although not as simple as one might expect. It is usually easier to reason at a more abstract level, using simpler languages, where one can be more confident in model correctness.

3.3 LLVM-Based Tools

The LLVM compiler infrastructure and LLVM Intermediate Representation (a.k.a., LLVM bitcode) provide an attractive and popular framework for program analysis and transform. We are developing LLVM-based tools that facilitate analysis:

- Many static analysis tools (e.g., Klee [5], llbmc [**?**], others) take LLVM bitcode as input. Unfortunately, producing bitcode from software source has become difficult, as the LLVM developers make arbitrary changes to their tool chain and to the bitcode syntax and semantics, without much regards to backward compatibility.

  Whole-Program LLVM (WLLVM) is an open-source project that help address these issues. The project provides tools to facilitate the production of LLVM bitcode for full C/C++ packages. In many cases, WLLVM transparently builds bitcode for C or C++ packages without any modification of the source. This works for both executables and libraries. SRI has made significant contributions to WLLVM and is now the main maintainer of the code.

- LLVM2SMT is a prototype that supports bounded model checking of LLVM code using SMT solvers. LLVM2SMT translates straight-line LLVM bitcode into an equivalent logical description in SMTLIB language. The translation uses the SMTLIB theory of fixed size bitvectors and arrays to enable bit-precise analysis (e.g., it correctly represents finite-precision arithmetic on 32bit integers). The resulting SMTLIB specification can then be checked using SMT solvers such as Yices, Z3, CVC4, etc. The main applications of this technology currently include test-generation and bug discovery using bounded model checking.

4. CONCLUSION

While developing the different tools described previously, we have made the following observations.

- It is inefficient to work directly at the binary level. Instruction sets are huge, not always well documented, and formalizing them precisely is tedious and error prone. This expensive step must be redone for each new architecture and instruction set.

- Working at the source code level is also inefficient as new programming languages emerge frequently, and existing ones change fast. Also, high-level languages have complex semantics such as implicit control flow, which are difficult to formalize.

- Intermediate representations such as LLVM or the JVM bytecode (or GCC's GIMPLE) are generally more stable and do not change as fast. They also allow us to cover a large set of high-level programming languages. LLVM, JVM, and GCC each support more than 20 languages. Formalizing the intermediate code representations is simpler than either high-level languages or low-level binary code. These intermediate representations are designed to support code transformations (such as compilation and optimization), and come with well-tested tools and infrastructure.

An assured, formal model of a simple enough IR is then key to efficiently develop reliable program transformations. IR is the right level of abstraction to build a formal model that we can trust and understand, while being expressive enough to specify useful transformations. Clearly, this is prerequisite to proving that our code transformations are correct, and in some case automatically synthesize transforms.

From such an IR model, we can also build static-analysis tools, symbolic-execution engines, and test-case generators. This allows us to validate and analyze transformations in cases where formal proofs are either too expensive or overkill.

We complement this IR model with a high-confidence decompilation from binaries to IR. Since all transformations happen at the IR level, ensuring that the decompilation is correct becomes a translation-validation problem. In other words, we prove that de- and re-compilation do not change the input/output behavior. We are planning on applying a variety of techniques including fuzzing, machine learning, and gray-box testing using our formal IR model.

This infrastructure can be reused beyond our immediate goal. It provides a set of tools and formalisms to quickly develop verified program transformations that can be used by developers of program-analysis tools, compiler engineers, and security experts.

Acknowledgement.
This work is funded in parts by AFRL contract No. FA8750-15-C-0010.

5. REFERENCES

[1] www.hex-rays.com.
[2] https://github.com/SRI-CSL/OpensDec.
[3] T.j.watson library for analysis (wala). http://wala.sf.net.
[4] S. Arlt, C. Rubio-González, P. Rümmer, M. Schäf, and N. Shankar. The gradual verifier. In *NFM*, 2014.
[5] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex system programs. In *USENIX Symposiom on Operating Systems Design and Implementation (OSDI 2008)*, pages 209–224, 2008.
[6] M. Co, J. Davidson, J. Hiser, J. Knight, A. Nguyen-Tuong, W. Weimer, J. Burket, G. Frazier, T. Frazier, B. Dutertre, I. Mason, and N. Shankar. Double helix and RAVEN: A system for cyber fault tolerance and recovery. In *Proceedings of the 11th Annual Cyber an Information Security Research Conference*, pages 17:1–17:4, 2016.
[7] B. Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.
[8] A. Gascón and A. Tiwari. Synudic: Synthesis using dual interpretation on components, 2015. http://www.csl.sri.com/users/tiwari/softwares/auto-crypto/.
[9] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.
[10] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Proc. ACM Conf. on Prgm. Lang. Desgn. and Impl. PLDI*, pages 62–73, 2011.
[11] T. Kahsai, J. A. Navas, D. Jovanović, and M. Schäf. *Finding Inconsistencies in Programs with Loops.* 2015.
[12] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. Jayhorn: A framework for verifying java programs. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, pages 352–358, 2016.
[13] S. Karaivanov, V. Raychev, and M. Vechev. Phrase-based statistical translation of programming languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, pages 173–184, New York, NY, USA, 2014. ACM.
[14] J. Ketema, J. Regehr, J. Taneja, P. Collingbourne, and R. Sasnauskas. Souper, 2016. https://github.com/google/souper.
[15] W. M. Khoo, A. Mycroft, and R. Anderson. Rendezvous: A search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 329–338, Piscataway, NJ, USA, 2013. IEEE Press.
[16] G. Malecha, A. Gehani, and N. Shankar. Automated software winnowing. In *SAC'15. Proceedings of 30th ACM Symposium on Applied Computing*, pages

1504–1511, 2015.

[17] T. McCarthy, P. Rümmer, and M. Schäf. Bixie: Finding and understanding inconsistent code. http://www.csl.sri.com/bixie-ws/, 2015.

[18] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK, UK, 2002. Springer-Verlag.

[19] C. Pacheco and M. D. Ernst. Randoop: Feedback-directed random testing for java. In *OOPSLA*, pages 815–816, New York, NY, USA, 2007. ACM.

[20] A. Tiwari, A. Gascon, and B. Dutertre. Program synthesis using dual interpretation. In *Proc. 25th Intl Conf on Automated Deduction, CADE*. Springer, 2015.

[21] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java Optimization Framework. In *CASCON*, 1999.

[22] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards optimization-safe systems: analyzing the impact of undefined behavior. In *SOSP*, 2013.