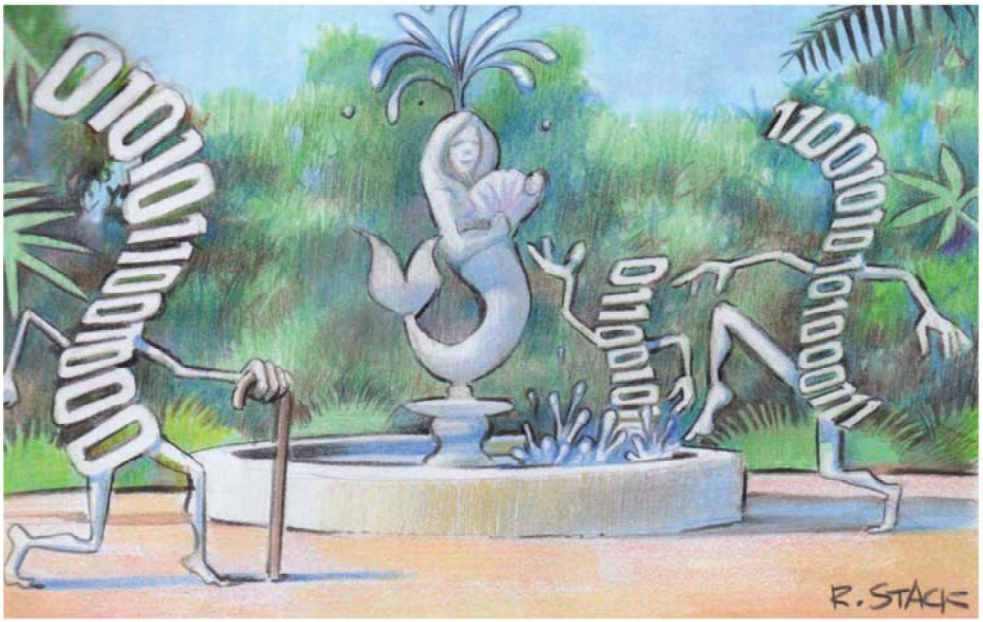
IT ALL DEPENDS

Editors: Mohamed Kaâniche, mohamed.kaaniche@laas.fr | Aad van Moorsel, aad.vanmoorsel@ncl.ac.uk

Binary Rejuvenation:

Applications and Challenges

Angelos Oikonomopoulos, Cristiano Giuffrida, Sanjay Rawat, and Herbert Bos | VU University Amsterdam



Software engineers have long performed *source code rejuvenation*—rewriting obsolete or outdated programming idioms to modern counterparts.[1] Taking inspiration from this practice, we suggest applying rejuvenation to the results of the compilation process. That is, we propose updating selected binary files, or *binary rejuvenation*.

It won't come as a surprise that the rejuvenation of binary executables is hard. Therefore, before discussing its challenges and possible solutions, we discuss why rejuvenation is useful.

Improved Health for Aging Code

Programming environments and frameworks often go to significant lengths to identify the optimal implementations and algorithms for their target workload and application. Common examples include "smart" versions of functions, such as `memcpy`, `strlen`, and `CRC32`, or specialized versions of graphics or cryptography algorithms. Similarly, individual programmers might import "faster" versions of platform-supplied functionality. The Internet is awash with posts detailing tricks of the trade.

It's doubtful that a project can consistently test and benchmark a custom implementation across its user base's range of CPU architectures and computing platforms. Even large frameworks for which abundant (corporate or cooperative) resources can be brought to bear must make tradeoffs to accommodate a bewilderingly diverse computing landscape.

The ever-evolving nature of the relevant hardware platforms exacerbates this situation. New microarchitectures, introduced at least every second or third year (https://www-ssl.intel.com/content/www/us/en/silicon-innovations/inteltick-tock-model-general.html), often bring specialized instructions that outperform software implementations of a software primitive (AES and `CRC32C` being two prominent instances). Moreover, cache sizes vary even within the same generation of CPU cores. The optimal implementation of a few years ago might now be inappropriate for modern processors or be rendered obsolete by a hardware primitive.

Besides performance, stale implementations might have unexpected security or reliability issues. New attacks on cryptographic operations exploit data-dependent branches to infer key material.[2] Similarly, internal hash functions that aren't collision resistant are an ongoing liability for software exposed to malicious input, for example, CVE-2011-4815, CVE-2011-4885, and CVE-2012-1150.

Modern OS distributions address this problem through shared libraries. In this way, when the platform is

Copublished by the IEEE Computer and Reliability Societies

updated, the application selects the implementation that's currently the best choice for a particular operation. Some programming environments take the extra step of dynamically selecting the implementation at symbol resolution time, accounting for the host CPU's microarchitecture.

Unfortunately, stale implementations still end up embedded in production binaries. It's common for organizations to rely on a proprietary binary that the vendor no longer updates because it isn't economically beneficial or because the vendor has gone out of business.

Although static linking is no longer the norm, frameworks or applications often import a specific external library version into their own source tree. They might do this because the project needs to make special modifications that would be unacceptable to a generic library version or, perhaps, because the upstream project hasn't taken care to preserve source or binary compatibility in its releases. When the range of targeted platforms is diverse enough, it's often impossible or impractical to rely on specific library versions being available, so static linking of dependencies becomes an attractive option. In every such instance, the resulting binary will end up with an embedded copy of external code.

Finally, portability considerations might also force a programmer to supply a fallback implementation of a primitive available on some, but not all, platforms targeted by the project. If the project must produce a generic binary, it has no choice but to embed its own implementation of that primitive. Even when developers attempt to publish separate binaries for a range of supported platforms, a function's fallback version might be selected accidentally because its inclusion is meant to be transparent during the build process.

Needless to say, clever implementations, such as those described, are sometimes used intentionally to substitute the platform-supplied versions—a decision that might have been appropriate when the binary was built.

The Problem

To replace one piece of machine code with another, the code should be relatively self-contained. With that in mind, we need to accomplish two tasks: identify the fragment of machine code to be replaced, and determine what to replace it with. This problem space has at least two dimensions: *equivalence* and *granularity*.

Software engineers have long performed source code rejuvenation. We suggest applying rejuvenation to the results of the compilation process.

For the first task, we offer a useful notion of what it means for a code fragment to be a legitimate target for replacement. Suppose we want to find a piece of code (in an assembler or a high-level language) embedded in a given binary. We must not only locate and prove this machine code's equivalence to our given fragment in the face of unknown compiler transformations but also ensure that our equivalence definition captures runtime effects and functional equivalence. Implementation details such as data-dependent jumps might also matter depending on the use case. In other words, the first task requires a narrow notion of equivalence.

For the second task, we must safely replace the identified code with a more appropriate version. Here, we can relax our notion of equivalence. Often, it will be enough if the replacement simulates (in the formal sense) the original. Other times, eliminating a fault will be the objective of the rejuvenation attempt. In most cases, the observable effects in terms of timing and asymptotic behavior will differ, which will necessitate human supervision because application requirements are rarely formalized.

For example, `bubblesort` and `mergesort` are functionally equivalent under virtually all definitions of the word and are fundamentally different in implementation. We could justify rejuvenating a binary by replacing the former with the latter, provided the application could tolerate the additional memory consumption.

The second dimension we're concerned with is the granularity at which we attempt binary rejuvenation. There are many possibilities, from wholesale replacement of old library versions to replacement of individual functions or parts of the callgraph.

Imagine rejuvenating inlined code that's intermingled with instructions performing unrelated operations. (Indeed, compiler built-ins are the exemplary instances of manually optimized binary code.) Replacing an inlined version of `CRC32` or a compiler-expanded or open-coded `memcpy` might very well be worthwhile. Yet it's a daunting task given that the fragment to be excised might consist of instructions dispersed over the control flow graph (CFG) of a larger function spanning multiple basic blocks and possibly participating in unrelated computations.

A Way Forward

A less lofty, more attainable, short-term goal is binary rejuvenation at the function level. Initially, we must be able to reliably identify function boundaries in well-formed binaries.

Luckily, there are already promising approaches to that problem.[3]

Many of the binary rejuvenation examples we've mentioned so far (`CRC32`, internal hash functions) are self-contained functions. We envision a database of function implementations, including metadata to determine each function's suitability for various hardware platforms, not unlike the way modern compilers select what code to expand a built-in to. For instance, if we locate a naive byte-by-byte version of `memcpy` in a binary intended to run in a newer Intel processor, we can replace it with a version using SSE (Streaming SIMD Extensions) instructions. Of course, we always prefer implementations that improve reliability or security.

Focusing on functions also makes the replacement procedure more tractable. The actual replacement of a function is straightforward: we emit the function's improved version and change all calls to target the new copy. Because the functions have no externally observable differences, remaining references to the old copy won't affect accuracy in programs that don't assume a low bound for execution time of the replaced function; they are simply missed optimization opportunities. That said, equivalent implementations might receive their inputs from and produce their outputs into different locations (registers or memory). Fortunately, when we identify equivalent code fragments, we can only do so under a specified correspondence of input and output locations between the two fragments, so the information is already there. Fragments with a variable number of input or output locations present an additional challenge.

Rejuvenation need not constrain itself to single functions. Any part of a callgraph can be identified and replaced; for instance, the `libressl` implementation of 3DES consists of the callgraph `DES_encrypt3 -> DES_encrypt2 -> DES_encrypt1`. We can safely replace a call to `DES_encrypt3` with a call to a completely different implementation of `3DES`, as long as we preserve the functions' original versions.

Precompilation could make binary rejuvenation a valuable tool in a deployment landscape where legacy binary code concerns are unavoidable.

Directions

How then do we identify an individual function implementation? Following are a few approaches.

Naive comparison. One way to determine equivalence is to look for functions that are byte-for-byte identical, in which case comparison is straightforward. The downside is that we must account for any differences in the object code due to different compilers, compiler versions, or compilation flags. Is that a viable tradeoff? It depends. Only a handful of compilers account for the vast majority of production binaries. The most active compiler project will make, at most, one minor release every few months. That makes for a few hundred relevant compiler versions in aggregate.

Code generation is most often determined by optimization flags; currently, five optimization levels (including -Os) are widely used. If we multiply these five levels by the number of distinct compiler versions, we end up with a few thousand. Given a C source file, precompiling it with all aforementioned combinations is quite manageable. The task is embarrassingly parallel, and the resulting machine code can be clustered to reduce the number of functions we have to search for.

However, modern compilers offer dozens of flags that affect code generation. Because these flags can be turned on and off individually, we'd run into combinatorial explosion if we tried to precompute every possible compilation result. However, if real-world development uses a relatively small number of flag combinations, this naive approach might suffice.

Beyond naivete. There are several ways to improve this algorithm. For one thing, we can abstract out differences in addresses (for position-dependent code) by using relocation information. For another, we can account for differences in the linear placement of basic blocks, first by comparing the CFG structure of the target function to (parts of) an existing binary program and then, if the structure matches, by doing a bytewise comparison of the corresponding basic blocks.

We could go further and lift the machine code to some generic intermediate representation before comparison, thus eliminating trivial mismatches due to different instruction selection. Along these lines, we could normalize instruction ordering and the computed subexpressions' structure. Liveness analysis and dead code elimination would further reduce false negatives.

Compiler transformations go far beyond these techniques. At some point, we might use a satisfiability modulo theory (SMT) solver to determine two basic blocks' equivalence. Specifically, SMT solvers are good at "solving" formulas in first-order logic, so we can express the

results of basic blocks as such formulas and then ask the SMT solver to see whether they can be satisfied. Even after establishing this, we'd need to ensure that the input and output locations of every basic block correspond consistently with the respective mappings for the function's other basic blocks.

Is this kind of normalization enough? Perhaps not. Recall that even simple compiler transformations such as partial redundancy elimination can change the CFG structure. Optimizations such as loop unrolling perform much more aggressive CFG modifications.

Our preliminary results suggest that for a set of representative built-in function implementations, minor versions of the same major GNU Compiler Collection (GCC) release (for example, 4.x.0 and 4.x.1) will create additional basic blocks and CFG edges that, approximately one-third of the time, can't be normalized away using the same optimization level. This happens when trying to recover the equivalence relationship of machine code generated from the same C source. Ignoring such cases, our normalization approach manages to show equivalence of the machine code functions for four out of every five pairs. When looking at function pairs compiled at different optimization levels or using different major GCC versions, our ability to recover equivalence deteriorates accordingly.

Should we continue down the normalization path? The answer isn't immediately obvious; expert rules on normalization are hard to develop, and compilers keep evolving. At some point, it might be worth employing approaches such as data-driven equivalence checking—identifying corresponding cut points between two dissimilar CFGs and proving the defined fragments' equivalence.[4]

Precompilation could amplify either approach to make binary rejuvenation a valuable tool in a deployment landscape where legacy binary code concerns are unavoidable. ■

References

1. P. Pirkelbauer, D. Dechev, and B. Stroustrup, "Source Code Rejuvenation Is Not Refactoring," *Proc. 36th Conf. Current Trends in Theory and Practice of Computer Science* (SOFSEM 10), LNCS 5901, 2010, pp. 639–650.
2. E. Tromer, D.A. Osvik, and A. Shamir, "Efficient Cache Attacks on AES, and Countermeasures," *J. Cryptology*, vol. 23, no. 2, 2010, pp. 37–71.
3. T. Bao et al., "BYTEWEIGHT: Learning to Recognize Functions in Binary Code," *Proc. 23rd USENIX Conf. Security Symp.* (SEC 14), 2014, pp. 845–860.
4. R. Sharma et al., "Data-Driven Equivalence Checking," *Proc. 2013 ACM SIGPLAN Int'l Conf. Object Oriented Programming Systems Languages and Applications* (OOPSLA 13), 2013, pp. 391–406.

Angelos Oikonomopoulos is a PhD student in computer systems at VU University Amsterdam. Contact him at a.oikonomopoulos@vu.nl.

Cristiano Giuffrida is an assistant professor of computer systems at VU University Amsterdam. Contact him at giuffrida@cs.vu.nl.

Sanjay Rawat is a postdoctoral researcher in computer systems at VU University Amsterdam. Contact him at s.rawat@vn.nl.

Herbert Bos is a professor of computer systems at VU University Amsterdam. Contact him at herbertb@cs.vu.nl.