Chimera: A Framework for Agile Repair of Live Applications

Garrett Tanzer
Harvard University

Abstract

As software services become increasingly complex without a commensurate improvement in the quality of the developer tools we use to manage them, the introduction of bugs becomes nigh-inevitable. When—despite our best efforts—one of these vulnerabilities is detected in a live product, our goal is now to mitigate its effect. But often this requires us to leave a susceptible system online, or else incur server downtime.

In this paper we present Chimera, a framework to dynamically splice managed code into primarily unmanaged applications. Chimera can replace functions in a C program with automatically generated safe versions that prevent classes of bugs resulting from undefined behavior, like buffer overflows, or integrate handwritten Python functions that address weaknesses stemming from logic errors, like the confused deputy problem. Once this stopgap is in place, developers can take the time to write a correct and performant native version of the function, then splice it in without requiring the server to restart.

The overhead of Chimera's live patching mechanism is negligible in practice, but our current implementation of C instrumentation (Google's AddressSanitizer) causes excessive slowdown in our test application—on the order of 20x. However, this can be improved with future work.

1 Introduction

Security holes are a reality, and our demand for performance-sensitive applications built upon immense codebases suggests that technical debt is here to stay. What is perhaps most concerning, given this model of the software development process, is how software updates are disincentivized. The recent Equifax breach was enabled by a late patch installation, allowing hackers to exploit a bug—CVE-2017-5638 in Apache Struts' Jakarta parser, which was fixed in March—in an attack two months later in May [9].

Even those who are more security-conscious face a dilemma. As the data sets that applications use grow faster than the ability for hard drives to serve it, the cost of restarting a server is prohibitively expensive: in the case of `memcached`, installing an update can incur several hours of downtime, meaning that services must be offered at a reduced capacity, or not at all [15, 10]. Companies like Facebook have developed techniques to reduce or amortize the cost of restarting these servers, but when a vulnerability is discovered, there is still an incentive to leave some susceptible servers live so that there is no period of complete outage. Because these services are so fine-tuned for performance, patches may not be immediately available after a vulnerability is discovered, because an inefficient initial patch would require a second server reboot once a more performant one is released.

Chimera is a platform to dynamically splice slower, safer code into a primarily performance-sensitive application to adapt as quickly as possible to these such conditions. Additional security guarantees such as bounds checking and type safety can be activated with the flick of a switch, and high-level, scripting languages can be substituted in as a stopgap until developers have had enough time to carefully craft a native patch.

The main contributions of this work are:

- the ability to enable stricter safety guarantees in an executing program on demand
- a seamless and dynamic interface between managed and unmanaged parts of the same program

Together, these two features allow performance to be maintained in the common case while granting adaptability.

The rest of this paper is organized as follows: Section 2 outlines how related work informs Chimera's design, Section 3 expands upon that design in more detail, Section 4 describes our prototype implementation of

Chimera, Section 5 evaluates that implementation, Section 6 identifies current weaknesses and areas for future improvement, and Section 7 summarizes our conclusions.

2 Related Work

Chimera synthesizes from several seemingly distinct areas.

2.1 Buffer Overflows

We identify three types of defenses against buffer overflow attacks, or more broadly attacks that result from incorrect memory behavior. First, some research aims to detect vulnerabilities at compile time. This often takes the form of constraint solvers, which evaluate programs on symbolic rather than concrete inputs and check whether their state follows certain predefined invariants [13, 6]. While effective, this type of static analysis has not been able to achieve full coverage, and so the concern of an uncaught vulnerability remains.

Second, there are tools that prevent buffer overflows from occurring at all. These uniformly apply bounds checking of some sort to C, like Baggy Bounds, which loosely bounds checks to the nearest power of two [4], or AddressSanitizer, which uses a compact form of shadow memory to catch many types of illegal memory accesses [19]. These strategies usually do not fully cover all possible attack vectors, and so their performance overhead is deemed unacceptable for production applications.

Third, there are runtimes engineered to prevent buffer overflow attacks from being successfully executed. This class of defenses arises in response to the expanded threat model of return oriented programming [17, 5], where an attacker strings together fragments of code already present in the address space, called *gadgets*, in order to accomplish an arbitrarily complicated computation. A common approach is to periodically randomize the program's code, disturbing any existing gadgets and making it exceedingly unlikely that a predefined series of them will do anything but crash [7, 21, 14].

2.2 Dynamic Updates

There are two parallel domains for dynamic software updates: kernel-level and user-level programs. Several kernel-level live patching schemes have been proposed and implemented for Linux, including `kpatch`, which adds a level of indirection to certain function calls, allowing small patches like security updates to be activated instantly [16], and KUP, which uses userspace checkpoint-and-restart to install even major releases in just a few seconds of downtime [12].

In user-space we see different approaches, like one used in the development of the server FlashEd, which uses proof-carrying code to verify that more substantial patches are indeed compatible with the live server [11], and fast Scuba restart, which uses shared memory mappings to transfer program state from an old to new version of software (albeit still with 2–3 minutes of downtime) [10].

Our strategy for Chimera is actually most similar to `kpatch`—using a level of indirection in function calls—but made more expressive by support for patches in different languages, as well as by the lack of the rigid interoperation and bookkeeping requirements inherent in kernel programming. Specifically, we recognize that programming in C takes more time than scripting languages, and we seek to minimize not only the time it takes to install a patch, but also the time it takes to write a compatible patch once a bug is discovered.

2.3 The Managed / Unmanaged Boundary

Several tools have been developed to interface between C and Python, but they remain inflexible and cumbersome. CPython, the reference implementation of Python, supports extending Python programs with C modules and embedding Python modules in C programs [2], but the arcane incantations needed to do so mean that numerical libraries are by far the biggest use of this feature, rather than regular applications. Cython, technically a superset of Python that is statically compiled yet still needs a separate Python interpreter to run, aims to bring Python closer to C [1], rather than to lend C Python's expressiveness. SWIG, the "Simplified Wrapper and Interface Generator," admirably glues a menagerie of languages together [3], but their connection is static. Chimera builds upon previous work by making the boundary between languages flexible: a caller and callee can switch dynamically from both being C, to being one of each, to both being Python and vice versa.

3 Design

In this section, we will survey the design of Chimera at a high level, then later delve into implementation details.

3.1 Live Patching

The key to Chimera's notion of dynamic updates is indirection, specifically through the Global Offset Table (GOT) and Procedure Linkage Table (PLT). Because dynamically linked libraries must necessarily be position-independent, global symbols referenced in the object file can only be resolved to specific addresses at link time—that is to say, when the library is loaded into an executing program. In the interim, these symbols reference positions in the GOT, whose position is fixed relative to the library's code. External functions also resolve to entries in the PLT, which are themselves function stubs that bounce execution to a section in the GOT.

When the dynamic linker opens a shared library, it resolves symbols in the GOT to fixed addresses from the program's global state. Therefore, subsequent accesses to external variables or functions are directed through the GOT and PLT before they reach their final destination. We can exploit this fact to update functions at runtime by overwriting the PLT of a library that has already been dynamically linked, immediately influencing the behavior of executing threads.

3.2 Managed C

Next, we consider what information needs to be available in order to splice in managed C code without warning. Chimera uses a two-pronged approach: a custom memory allocator intercepts and annotates heap allocations, while DWARF debugging data recovers type information about variables on the stack. When a memory-safe C function is invoked, it must have a function preamble that looks up and validates variable bounds and types before it begins to do work. We distinguish code which uses the custom memory allocator but not bounds checking from that which uses both by the terms "uninstrumented" and "instrumented" code, referring to the bounds checking operations themselves.

This is sufficient to enforce access to valid memory regions, but not necessarily correctly typed access. For stronger guarantees about the behavior of our managed code, we need to restrict the ability of uninstrumented C code to transform types: specifically, Chimera prohibits casting between numerical values and pointers, including as part of unions. We do, however, necessarily admit exceptions, like the `void *arg` parameter in `pthread_create`.

This is particularly subtle in `struct`s. For example:

```
struct ufoo {           struct foo {
    uint8_t a;             int8_t a;
    uint32_t b;            int32_t b;
    struct ufoo *next;     struct foo *next;
};                      };
```

`struct ufoo *` and `struct foo *` can be cast between each other because their struct definitions preserve pointer locations, but they cannot be cast to a `char *`, which points to just a numerical value. We will see later that this also allows Python's garbage collector to peek outside of the managed world and track objects that have entered unmanaged C code.

3.3 Python

As we know that it is possible to build a Python interpreter, the interesting considerations for Python in the context of Chimera come at the boundary between the managed and unmanaged world—that is to say, precisely how objects pass between the two sides. We have identified two main areas that bear examination.

3.3.1 Trampolines

Whereas managed C code can be spliced into the PLT with ease, so long as the relevant function preambles are present, we must be more cautious with Python code because it is not executed natively. When a C function is directed through the PLT towards a method that is implemented in Python, it must not only be sanity checked in the same way as managed C, but its parameters must be coerced into types that a reasonable Python programmer would expect—including transforming pointers into references to the objects or arrays of objects that they represent. Then, upon exiting the function, we must unwrap those types to reach meaningful C variable state before we return control to the caller. We call these bouncing-off-points "trampolines," and so we see that pointers to Python functions themselves ought not be stored in the GOT, but rather pointers to trampolines that invoke the Python interpreter.

The interpreter must also be able to serve as a trampoline if the Python function calls another that is written in C, and would ideally be able to short circuit the process when calling another function defined in Python, to avoid redundant wrapping and unwrapping. This optimization, along with how to efficiently store Python objects to reduce the cost of type coercion, are questions to be addressed during implementation.

3.3.2 Sticky Garbage Collection

Beyond how we will transform objects that cross the managed/unmanaged boundary into an accessible state, we must also examine how to control the lifetime of data that is allocated on one side and crosses over into the other. This is complicated by the fact that C has synchronous, manual memory semantics, whereas Python has asynchronous garbage collection. We call our solution is "sticky garbage collection."

The "sticky" in sticky garbage collection refers to the idea that any object that is touched by Python is agglomerated into its mass of objects—this is necessitated by the fact that a C object passed by reference into Python could be added into some persistent data structure. When `free` is called on a pointer, it will behave in one of two ways: if that object has been tainted by the managed world, `free`

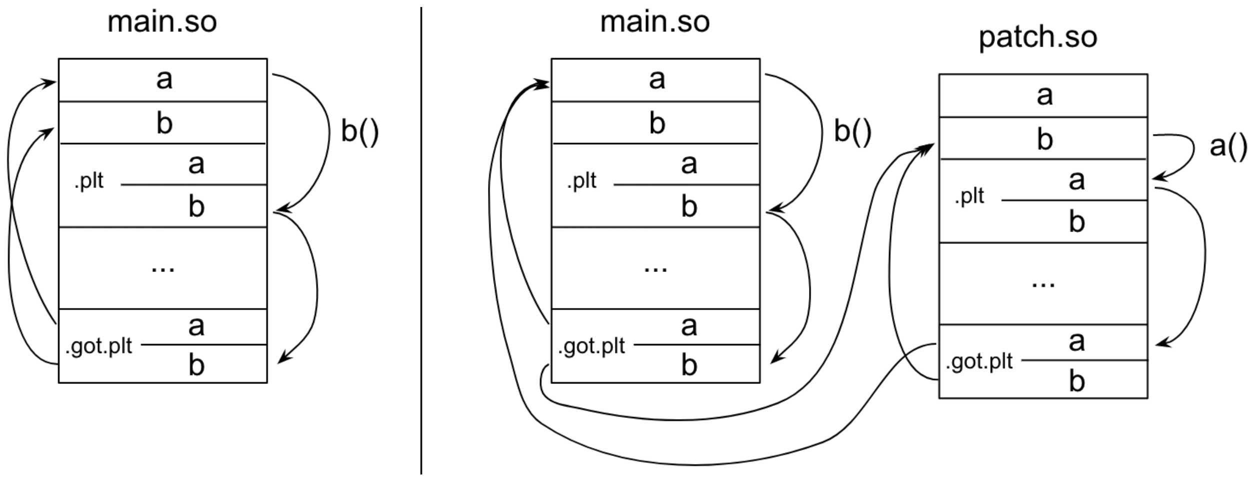

Figure 1: PLT and GOT of `main.so` before and after splicing in `b()` from `patch.so`.

will zero the pointer's location in memory, effectively removing a reference to the object (in order to do this, we need to add `&ptr` as a parameter to `free`). If it has not touched the managed world, `free` will behave like a normal, synchronous free.

Thankfully, we already have the information we need to allow the Python garbage collector to traverse the unmanaged world—not conservatively, but exactly. We discussed above how bounds and type information will be tracked to facilitate the boundary with managed C, and we can use this same information to find all C-reachable objects. Whether we need to pause running C threads in order to check for object reachability—but not reorganize any C state to avoid fragmentation—is another question for implementation.

4 Implementation

We created a prototype in Linux with the first two components of Chimera: live patching and automatic generation of managed C. Our high-level goal was to use existing compiler flags as much as possible, rather than to add customized LLVM passes.

4.1 Live Patching

Chimera's architecture is as follows: a base program, `chimera`, spawns a daemon thread that waits on a Unix domain socket (UDS) to be notified of patches. To ensure that all function calls in the user program are directed through the PLT, we compile the desired base executable as a shared object, then pass its file name as a command-line parameter to `chimera`. We open the base library with `dlopen`, locate its `main` function with `dlsym`, then launch the program itself.

When we want to install a patch, we compile it into a shared library, then use `objcopy` to weaken all of its global symbols so that static variables and function pointers will resolve consistently to their locations in the base library through the PLT—even for local or recursive calls. We then send the name of the library and desired function over the UDS; at the other end, the Chimera daemon opens the library and dynamically links it into the process. Next, we traverse the `struct link_map` associated with the base ELF file to reach the relevant `.got.plt` entry, and change the pointer linked to our desired function with a single, atomic write [20]. We repeat this short process for as many patches as are currently linked in, since each library has its own GOT. See Figure 1 for a concrete example of the resulting structure.

4.2 Managed C

Instead of translating the machine code for a particular function into a safer version at runtime, we elected to generate an entire second binary at compile time, when C semantics are still intact. We currently use AddressSanitizer, which is built into `gcc` and `clang`, to create this instrumented binary. "Unmanaged" files are compiled with AddressSanitizer on but instrumentation off, which means that they still use the tool's intercepted memory allocators (allowing objects allocated in unmanaged code to be sanity checked in managed code), but do not bounds check their own operations. This memory-safe version can be loaded by Chimera like any other type of patch, and code can be swapped out at function granularity. While this does effectively increase binary size, only the functions in active use should be left resident in memory by the OS's virtual memory system, and for the types of applications we are concerned with, binary size should be small compared to the working set.

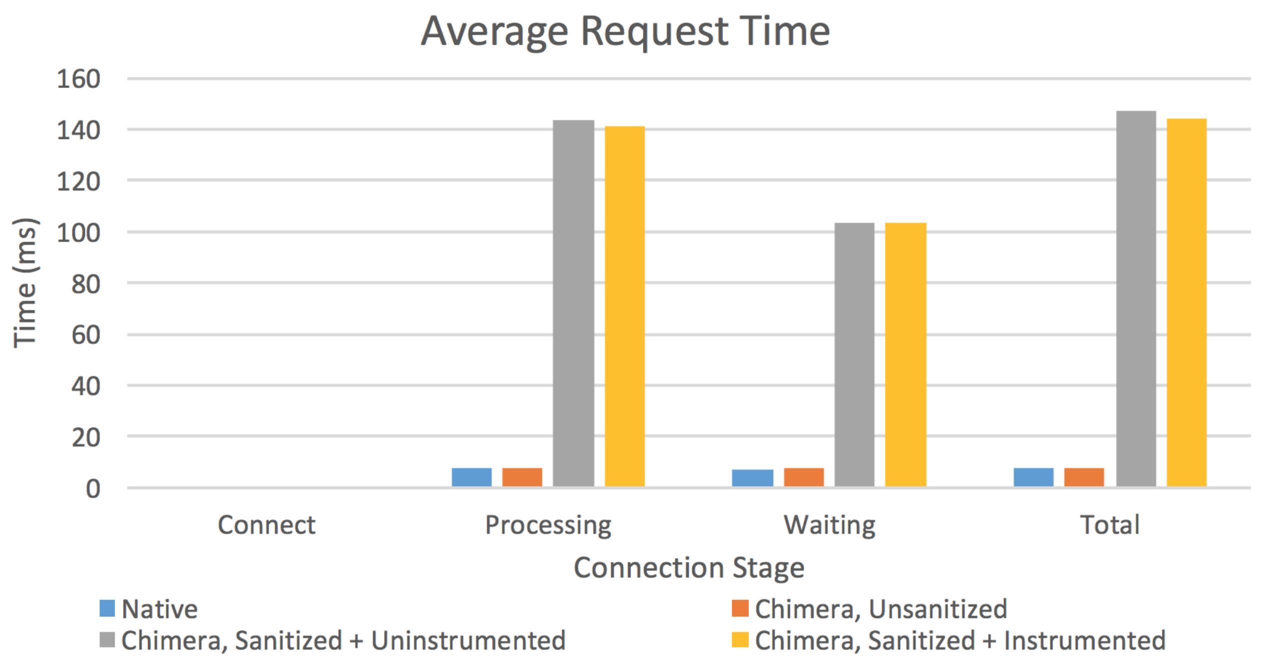

Figure 2: Latency over 5000 HTTP GET requests to different configurations of the toy server, with at most 350 concurrent connections.

5 Evaluation

In this section, we evaluate Chimera on a toy web server [18]. No changes to the server's source code were necessary except the addition of a `gcc` attribute to each function signature, but this can be automated in the future. We compared the behavior of four configurations:

- a native, statically linked version
- a dynamically patchable, unsanitized version
- a dynamically patchable, sanitized, but uninstrumented version
- a dynamically patchable, sanitized, and instrumented version

5.1 Correctness

We deliberately introduced a buffer overflow vulnerability into our test server, then attempted to exploit it on the four configurations. As expected, the attack succeeded on the native version of the executable. It also succeeded on the unsanitized Chimera version, but we were able to patch the hole by splicing in a secure version of the susceptible function. The sanitized but uninstrumented version of the server unexpectedly detected the attack, and we later discovered that this is because AddressSanitizer calls instrumented versions of `libc` functions—in this case `strcpy`—even from uninstrumented programs. We modified the vulnerability to occur in user-written code, rather than library code, and it then evaded detection. The fully instrumented version successfully detected both attacks.

5.2 Performance

We used ApacheBench [8] to load test our server configurations locally. The test was run in an Ubuntu 16.04.3 VM with 4 cores and 4 GB of RAM using VMWare Fusion 8.5.8, on a MacBook Pro with an i7-4870HQ CPU @ 2.50 GHz and 16 GB of RAM. Our test consisted of 5000 total requests, with a maximum of 350 concurrent connections, and a timeout of 30 seconds (which was never reached). The first two configurations had a median 7ms response time with a standard deviation of 4ms, while the two sanitized versions had a median 145ms response time and a standard deviation of 3500ms, where the distribution was heavily right skewed.

We attribute this overhead in the sanitized but uninstrumented to instrumented `libc` functions, as mentioned above, though we also note that this 20x slowdown is far greater than the 2x average in the published AddressSanitizer benchmarks [19]. We also tested a sanitized native version without support for dynamic patching to ensure that the slowdown is not an interaction between AddressSanitizer and shared libraries or position-independent code, and our results were virtually the same.

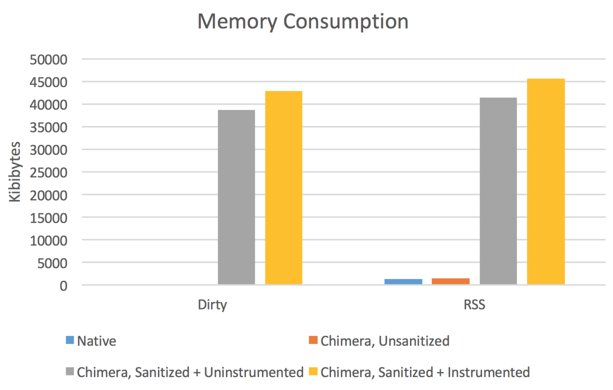

Figure 3: Dirty bytes and resident set size of different server configurations.

5.3 Memory Usage

We also used `pmap` to inspect the memory mappings of our configurations. AddressSanitizer preemptively maps its entire shadow region into address space, which artificially inflates the "total kilobytes" metric; so, we only present the number of dirtied KiB and the resident set size of the process.

The unsanitized version of Chimera has 72% more dirty bytes and a 15% larger resident set size than the native version of the toy server, which can mostly be attributed to the fixed-size code that installs patches, so this cost is amortized as the application itself increases in scope. Meanwhile, we see again that AddressSanitizer has an enormous footprint, though this also decreases proportionally as the application grows.

6 Limitations and Future Work

AddressSanitizer: The performance of sanitized C functions is clearly unsuitable for Chimera's intended use case. Although AddressSanitizer does support uninstrumented functions, this is not the project's focus; bounds checking is only disabled in the user-defined part of the function, not any `libc` calls that it may make, so applications reliant on string or memory operations (e.g. web servers) see nearly the same overhead as full instrumentation. It may be possible to modify AddressSanitizer to link both instrumented and uninstrumented versions of relevant library functions for use by the appropriate functions, but a new approach optimized for low bookkeeping overhead in the uninstrumented case would likely have better performance.

Python: In order to fully evaluate Chimera, we must also implement and benchmark the Python interface described above. This is a nontrivial but relatively straightforward endeavor.

dll Injection using ptrace: Currently, Chimera opens patches as dynamic libraries and interposes on the PLT from a daemon thread, but it is arguably dangerous to have such powerful code in the same address space as the user application. If the program fell victim to a ROP attack, the attacker would have access to gadgets that can introduce new executable code into the process and redirect global control flow, much like exploits that overwrite the function pointers in C++'s `vtables`. Instead, we can use a sentinel process and `ptrace` to inject shared objects into the address space of the Chimera application, isolating the code responsible for doing so. In this way, we can reduce the number of gadgets we expose while decreasing binary size. This also allows us to avoid the difficulty of propagating the Chimera daemon across forked processes.

Rigorous Benchmarking: Once we resolve the performance issues with uninstrumented AddressSanitizer, we plan to reevaluate Chimera by building and stress testing Apache HTTP Server, as well as the SPEC CPU2006 test suite.

7 Conclusion

In this paper we presented Chimera, a runtime that features near-native performance (okay—not yet, but once it's fixed) and can enforce additional security guarantees on demand, at the cost of temporary performance degradation. Seamless splicing of high-level Python into unmanaged C code enables rapid iteration on live applications—whether security updates or feature upgrades—without the need for downtime, minimizing not only the time from patch release to installation, but also bug discovery to patch release.

Acknowledgements

I would like to thank Prof. James Mickens for his guidance and eternal wisdom, as well as for his awareness of ApacheBench. I'd also like to thank Prof. Eddie Kohler for helping me figure out what to whisper in `gcc`'s ear to get AddressSanitizer to behave.

References

[1] C-extensions for python.

[2] Extending and embedding the python interpreter.

[3] Simplified wrapper and interface generator.

[4] AKRITIDIS, P., COSTA, M., CASTRO, M., AND HAND, S. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. *Proceedings of the 18th USENIX Security Symposium* (2009).

[5] Bittau, A., Belay, A., Mashtizadeh, A., Mazières, D., and Boneh, D. Hacking blind. *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (2014).

[6] Cadar, C., Dunbar, D., and Engler, D. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proceedings of the 8th USENIX Symposium on OSDI* (2008).

[7] David, L., Liebchen, C., Sadeghi, A.-R., Snow, K. Z., and Monrose, F. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. *Proceedings of the NDSS Symposium 2015* (2015).

[8] Foundation, A. S. Apache http server benchmarking tool.

[9] Fox-Brewster, T. How hackers broke equifax: Exploiting a patchable vulnerability.

[10] Goel, A., Chopra, B., Gerea, C., Mátáni, D., Metzler, J., Haq, F. U., and Wiener, J. Fast database restarts at facebook. *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (2014), 541–549.

[11] Hicks, M., Moore, J. T., and Nettles, S. Dynamic software updating. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (2001), 13–23.

[12] Kashyap, S., Min, C., Lee, B., Kim, T., and Emelyanov, P. Instant os updates via userspace checkpoint-and-restart. *Proceedings of the 2016 USENIX Annual Technical Conference* (2016), 605–619.

[13] Larochelle, D., and Evans, D. Statically detecting likely buffer overflow vulnerabilities. *Proceedings of the 2001 USENIX Security Symposium* (2001).

[14] Morton, M., Koo, H., Li, F., Snow, K. Z., Polychronakis, M., and Monrose, F. Defeating zombie gadgets by re-randomizing code upon disclosure. *Proceedings of ESSoS 2017* (2017), 143–160.

[15] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling memcache at facebook. *Proceedings of the 10th Symposium on Networked Systems Design and Implementation* (2013), 385–398.

[16] Poimboeuf, J., and Jennings, S. Introducing kpatch: Dynamic kernel patching.

[17] Roemer, R., Buchanan, E., Shacham, H., and Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security 15*, 1 (2012), 2:1–2:34.

[18] Sanchez, O. Simple webserver (miniweb).

[19] Serebryany, K., Bruening, D., Potapenko, A., and Vyukov, D. Addresssanitizer: A fast address sanity checker. *Proceedings of the 2012 USENIX Annual Technical Conference* (2012), 309–318.

[20] Takehiro, K. Plthook.

[21] Williams-King, D., Gobieski, G., Williams-King, K., Blake, J. P., Yuan, X., Colp, P., Zheng, M., Kemerlis, V. P., Yang, J., and Aiello, W. Shuffler: Fast and deployable continuous code re-randomization. *Proceedings of the 12th USENIX Symposium on OSDI* (2016).