BlankIt Library Debloating

Getting What You Want Instead of Cutting What You Don't

Chris Porter*

Georgia Institute of Technology
Atlanta, GA, USA
porter@gatech.edu

Prithayan Barua

Georgia Institute of Technology
Atlanta, GA, USA
prithayan@gatech.edu

Girish Mururu*

Georgia Institute of Technology
Atlanta, GA, USA
girishmururu@gatech.edu

Santosh Pande

Georgia Institute of Technology
Atlanta, GA, USA
santosh.pande@cc.gatech.edu

Abstract

Modern software systems make extensive use of libraries derived from C and C++. Because of the lack of memory safety in these languages, however, the libraries may suffer from vulnerabilities, which can expose the applications to potential attacks. For example, a very large number of return-oriented programming gadgets exist in glibc that allow stitching together semantically valid but malicious Turing-complete and -incomplete programs. While CVEs get discovered and often patched and remedied, such gadgets serve as building blocks of future undiscovered attacks, opening an ever-growing set of possibilities for generating malicious programs. Thus, significant reduction in the quantity and expressiveness (utility) of such gadgets for libraries is an important problem.

In this work, we propose a new approach for handling an application's library functions that focuses on the principle of "getting only what you want." This is a significant departure from the current approaches that focus on "cutting what is unwanted." Our approach focuses on activating/deactivating library functions on demand in order to reduce the dynamically linked code surface, so that the possibilities of constructing malicious programs diminishes substantially. The key idea is to load only the set of library functions that will be used at each library call site within the application at runtime. This approach of demand-driven loading relies on an input-aware oracle that predicts a near-exact set of library

functions needed at a given call site during the execution. The predicted functions are loaded just in time and unloaded on return.

We present a decision-tree based predictor, which acts as an oracle, and an optimized runtime system, which works directly with library binaries like GNU libc and libstdc++. We show that on average, the proposed scheme cuts the exposed code surface of libraries by 97.2%, reduces ROP gadgets present in linked libraries by 97.9%, achieves a prediction accuracy in most cases of at least 97%, and adds a runtime overhead of 18% on all libraries (16% for glibc, 2% for others) across all benchmarks of SPEC 2006. Further, we demonstrate BlankIt on two real-world applications, sshd and nginx, with a high amount of debloating and low overheads.

CCS Concepts: • Security and privacy → Software security engineering.

Keywords: software debloating, program security

ACM Reference Format:

Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. 2020. BlankIt Library Debloating: Getting What You Want Instead of Cutting What You Don't. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20)*, June 15–20, 2020, London, UK. ACM, New York, NY, USA, 17 pages. <https://doi.org/10.1145/3385412.3386017>

1 Introduction

Modern software relies heavily on libraries that are often built for supporting a large amount of functionality. In a given application, however, only a small amount of such functionality may get used. For example, programmers leverage Android libraries, lean on machine learning and AI tools, and build on top of web frameworks to improve software development productivity [3, 9, 19, 43, 59]. Although these frameworks can be dauntingly large, it is normal to use only a small subset of the APIs. A recent study [49] shows in userspace programs that ship with Ubuntu Desktop 16.04, only about 10% of the shared library functions are used.

*Both authors contributed equally to the paper

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

PLDI '20, June 15–20, 2020, London, UK

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-7613-6/20/06...\$15.00

<https://doi.org/10.1145/3385412.3386017>

For performance reasons, these underlying libraries are inevitably written in C/C++. The lack of memory safety in C/C++ leads to a large number of exploits that expose the applications or frameworks that use the libraries, in turn leaving them vulnerable. One such library that forms the core of C/C++ libraries and applications is the GNU version of libc, or “glibc” [23]. Glibc also acts as a basic building block of other libraries and their higher-level APIs in GNU/Linux systems. Unfortunately, the list of vulnerabilities and exploits in glibc keeps growing, with 108 known and published vulnerabilities at the time of this writing [13], of which nine were published just last year. Moreover, glibc is a mine of return-oriented programming [54] (ROP) gadgets that can be stitched together to create semantically valid and malicious attacks. Although known CVEs are patched and remedied, new/future attacks are built over these gadgets. Hence, reducing the quantity and expressiveness of such gadgets for libraries like glibc is an important problem for which numerous strategies are still being invented.

1.1 Current Defense Mechanisms

Hardware and software defense mechanisms have been developed to detect and mitigate the attacks, but even state of the art techniques have limitations. For instance, address-space layout randomization [47] (ASLR) and data execution prevention [2] (DEP) cannot protect against advanced control-flow hijack techniques like ROP and jump-oriented programming [7] (JOP). To defend against the exploitation of these advanced vulnerabilities, specific and general mechanisms have been invented. One such well-known defense mechanism that has been researched extensively is control-flow integrity (CFI) [1, 6, 12, 22, 26, 30, 35, 70], a mechanism that attempts to restrict the program execution to legal paths only.

While coarse-grained CFI such as bin-CFI (a binary-level CFI technique) [72] restricts jumps to ROP and JOP gadgets at addresses other than those of functions, it over-approximates the legal execution paths possible through calls and returns. This allows for malicious program paths and thus overlooks certain attacks. TypeArmor [62] tries to improve on bin-CFI by reducing the over-approximation of forward edges. To further reduce the approximation of the allowed set of calls from a call site and to maintain legal returns, a number of fine-grained CFI techniques such as π CFI [42] and MCFI [41] leverage the source code of libraries. However, computing a fully-precise CFI is an intractable problem because of the inherent limitations in pointer analysis [18], irregular control flow in the presence of `setjmp` and `longjmp`, or an inherent inability to distinguish between a legal and illegal dynamic control flow base with regard to the current program input. μ CFI [26] leverages Intel Processor Trace and static analysis to enforce its unique target code property for indirect control-flow transfers. It broadens the scope of CFI by handling non-control data corruption that can alter function pointers.

Memory safety is a defense mechanism that prevents memory corruption by verifying the correctness of memory operations. While memory safety techniques like Soft-bound+CETS [38] can thwart almost all control-flow hijacking attacks, they incur overheads $\geq 2x$. Due to the large overhead, it is impractical to deploy it online. Code pointer integrity (CPI) [27] is a defense mechanism that uses memory safety only on pointers that can directly or indirectly modify code (control-flow), thus lowering the overheads of a full memory safety mechanism. Control-flow bending (CFB) via generalization of non-control data attacks shows that a fully precise, static CFI approach cannot defend against many non-control data attacks, and similarly CPI suffers from such attacks. CFB demonstrates that with the availability of a dispatcher function, commonly available in glibc, a CFI mechanism can be fooled to successfully carry out an attack. CFI also does not handle certain types of privilege escalation attacks [11, 73]. Due to this, techniques like CFI and CPI still need to be complemented with defense mechanisms for these specific attacks that fall outside their scope. Moreover, as defenses strengthen, new attacks are revealed that undermine the current best practices of defense mechanisms. Due to the sophistication of attack construction tools, the number of newly discovered attacks are growing at a much faster rate, which may lead to a losing battle [60].

1.2 Debloating

To reduce the effects of such unseen attacks, a promising way forward is program debloating [31], that is, downsizing the amount of code available for constructing an attack gadget. Unwanted or dead code is removed from the application or library. Current compiler-based link-time optimizations statically determine a set of functions to be removed from the linked library using a combined analysis of unreachable functions, global constant propagation, and inter-procedural evaluation of branch outcomes using the fixed points of unreachable code and (conditionally propagated) constants [14, 29, 57, 58]. Software debloating at call sites that use function pointers faces the same limitation of over-approximation seen in CFI, allowing construction of gadgets from available functions. Although static program debloating is a positive step towards thwarting unknown possibilities of attacks and towards reducing attack surface area, it is quite ineffective due to the limitations of static analyzability of programs. A lot of code remains linked.

Piece-wise compilation [49] improves purely static techniques by collecting address-taken functions of each module. At load time, it brings in only those functions that are deemed required by the program. While it reduces the number of functions at an attacker’s disposal, dynamically unused functions are still loaded. It debloats between 60% to 86% for musl libc, leaving a number of ROP and JOP gadgets linked to the program. Further, most software debloating mechanisms require *source code analysis* and hence omit important,

widely used libraries such as glibc, which is notorious for extremely complex control flow, poses a huge obstacle for inter-procedural analysis of reasonable sophistication (static or dynamic), and cannot even be compiled with clang/LLVM at present. In fact, the current glibc is not a realistic target for static linking. It uses dlopen to pull in additional libraries that *themselves* depend on glibc. Thus, even if one could statically link the main application with glibc, at runtime inadvertently another copy of glibc is dynamically loaded and linked.

1.3 BlankIt: "Get what you want"

Current security debloating approaches can be classified into two categories: conservative ones that remove only those functions whose removal is deemed safe under the soundness assumption (e.g. piece-wise compilation) [49], or aggressive ones, which break soundness and can lead to crashes (e.g. Razor or Chisel) [24, 50]. Conservative approaches leave a considerable amount of unwanted code linked to an application, significantly degrading security. On the other hand, aggressive but unsound approaches suffer from usability issues (such as crashes, and/or incorrect execution), making such schemes impractical. Guaranteeing soundness requires verification, which is intractable for any reasonably sized programs.

In response to the limitations above, we introduce BlankIt. **BlankIt is designed to get only the functions required at a library call site in the current dynamic calling context, rather than cut the statically unwanted functions.** This fundamental departure yields a high amount of debloating (almost 97%). BlankIt relies on the following core ideas:

- It loads only the set of needed/predicted functions from the library at a call site on demand at runtime. The predictor, a decision tree-based learner, acts as an input-sensitive oracle and provides the list of required functions. The other functions in the library have been blanked out with invalid instructions.
- Library functions run under the supervision of probes, which fire before each function's invocation. If more functions are needed for execution, they are loaded on demand.
- When the library execution returns to the call site, the complete call chain of loaded functions is purged.
- When a probe detects a misprediction, a mitigation mechanism is triggered. We list several schemes here and include an evaluation of a parallel audit process that reviews the alarm.

Differences between BlankIt and other debloating schemes are illustrated in Figure 1. BlankIt's function set is often a small subset of the functions needed by a program. This subset dynamically shrinks and grows, mostly staying within the innermost boundary of the needed functions. The prediction accuracy of BlankIt is very high, but if BlankIt under-predicts

the set of needed functions, it is able to load them in a safe manner. In a very small number of occurrences, BlankIt over-predicts by a small degree. Due to these reasons, BlankIt achieves very aggressive debloating, diminishing the attack construction surface in libraries to near-zero while guaranteeing a very high amount of usability of continuity of execution (without crashes or incorrect execution). We show in Sections 5 and 6 that BlankIt has a very high security benefit to avoid currently known CVEs and vulnerabilities.

Figure 1. BlankIt compared with other debloating mechanisms

1.4 Overcoming CFI Limitations

BlankIt can be combined with CFI and should not be seen as a replacement. Though there is some functional overlap, BlankIt can complement CFI and handle scenarios that CFI would otherwise miss. For instance, there are environments where many CFI techniques do not penetrate, such as glibc. BlankIt's predictive approach can identify valid jumps for set jmp/long jmp easily compared to CFI mechanisms. And most crucially, the prediction approach enables BlankIt to tackle non-control data attacks, as well, which is outside the scope of all CFI implementations and CPI [26].

To illustrate, consider the non-control data attack from [73] shown in Listing 1. A user's credentials are verified with an input key in function service, and the result is assigned to the variable user. The user is frequently checked to provide access to certain operations. Between checks, however, the function interacts with the program user. Input some input can be manipulated by the program user, causing a buffer overflow at line 7, and leading to superuser access. This attack is not handled by CPI or the latest CFI (π CFI, μ CFI), but in the case of BlankIt, the call_super_user function is not predicted, and the attack is detected in the audit and mitigation phase that checks for violations. Note that CFI combined with a software-debloating mechanism like piece-wise compilation [49] also cannot handle this attack. Another of BlankIt's advantages is that it is a binary defense mechanism and works on glibc, unlike most state-of-the-art defense

mechanisms like CPI, π CFI, or CETS. BlankIt is limited, however, to handling such attacks only when a function call is involved, since that is the prediction granularity it currently operates on.

```

1 void service(ulong key) {
2     char str[SIZE], user[SIZE]; ...
3     verify_user(user, key);
4     if(strncmp(user, "admin", 5)){ ... }
5     else{ ... }
6     strcpy(str, someinput);
7     if(strncmp(user, "admin", 5))
8         call_super_user();
9     else
10        call_normal_user();
11 }
12 void access_service(char pwd[20]) {
13     int auth = verify_pwd(pwd);
14     if(auth)
15 B1:    key = calculate_key_su(pwd);
16     else
17 B2:    key = calculate_key_nm(pwd);
18     service(key);
19 }

```

Listing 1. Attack missed in CPI and μ CFI but caught in BlankIt

1.5 Threat Model

We assume that the program is not self-modifying, and an attacker can read/write the data section and read/execute the code section of a vulnerable program. We assume that the application source, LLVM compiler that instruments the application, the dynamic BlankIt runtime, and the underlying hardware, OS, and loader are not compromised. Prediction calls to BlankIt from user applications are protected against any argument tampering using pass-by-register semantics; further, these arguments are guaranteed to fall within a compile-time known prediction set. Application code pages are $W \oplus X$ (writable XOR executable), and BlankIt maintains this property between copying and blanking operations to guard against code injection (and potentially probe insertion). All external libraries can be a source of threat, and any such external call must be protected. Untrusted application code can be modularized, however, into shared libraries and protected with BlankIt, just like untrusted third-party libraries. We make no assumptions on obfuscated code or hand-written binaries or any other calling conventions.

We assume a protected runtime environment for BlankIt, independent of its current prototype in Pin. These solutions have already been explored, e.g. CPI's in-process isolation [27], and a BlankIt runtime can leverage hardware segmentation or software fault isolation [28] that guards runtime against attacks such as [17]. This prevents attackers from jumping into BlankIt's runtime probes. Repeatedly calling BlankIt's copy and blank functions does not change the

state of the application nor increase dynamically linked functions. Influencing or somehow steering the decision trees is impossible. The decision trees are embedded and compiled into application code and use read-only, scalar arguments. BlankIt-inserted trampolines cannot be triggered by an attacker either. Such an arbitrary trampoline would be caught in BlankIt the same way any arbitrary jump is caught (due to illegal control flow). Note that some of these attacks are also caught by CFI mechanisms with which BlankIt can be augmented.

1.6 Limitations

Like other debloating techniques, vulnerabilities against loaded instructions are not protected by BlankIt. For example, if an attack exploits a buffer overflow inside of a loaded library call chain, and then uses only the gadgets within that call chain, then the attack will go unnoticed by BlankIt. Only by landing in unloaded memory, or by triggering a mispredict, does BlankIt thwart attacks.

1.7 Contributions

The following are the contributions of this work:

1. A framework for dynamically loading and unloading library functions on demand at binary level.
2. A prediction mechanism within that framework for predicting the required library functions to be loaded at each call site, thereby reducing over-approximation.
3. An alarm and audit technique that on misprediction runs a memory safety check on the library call.
4. An evaluation of our framework using SPEC CPU 2006 benchmarks and its supporting libraries (glibc, libm, libgcc, and libstdc++), some of which are not handled in other approaches that depend on library source; and an evaluation on sshd and nginx, with treatment of over 10 more libraries and support for multiprocessing and multithreading.

These result in the following security implications:

1. The set of active dynamically linked functions is reduced to protect against future, unforeseen exploits that mainly rely on code reuse and replay-based attack construction, and we provide reasonable metrics and results for the same.
2. The glibc library is addressed. It is notorious for vulnerabilities and not tackled by most state-of-the-art defense techniques. It is ubiquitous and a significantly higher-impact library than alternatives like musl [36].
3. Certain non-control data attacks are thwarted (as seen in the example in Listing 1 or the nginx case in Section 6.4), which fall outside the scope of CFI and CPI.

2 BlankIt's Overall Framework

BlankIt is a compiler-aided runtime environment for demand-driven loading of library functions. At runtime, BlankIt is

initialized at program-start, when libraries are first brought into memory, and BlankIt immediately does the following for every function: It inserts a probe at the function entry, copies the function code after the probe into a safe read-only memory, and wipes (blanks) out all the code after the inserted probe. Later during runtime, BlankIt loads library functions on demand.

The application itself is instrumented beforehand. A decision tree learnt through off-line profiling is instrumented for every library call site in the application by the compiler (LLVM) pass. At runtime, the decision tree simply outputs the set of functions within the library that it expects to invoke at a given call site under the current dynamic calling context. That context consists of call site ID, reverse dominance frontier (RDF) of static single assignment (SSA) values of arguments to the call, and the library function’s arguments themselves. When the application calls a library function, the probe (inserted when the library was imported) gets invoked. This probe checks if the functions called are as predicted. If they are, execution continues in the loaded functions. If, however, the call is not through legitimate means (non-predicted and not legal) then an attack is detected. Any attempt to bypass the probe results in a fault or crash, because the code section has been blanked. Finally, when the control returns to the application, the functions that were copied are reset back to a blanked state. This purging is what minimizes dynamically linked functions.

A simplified diagram of BlankIt’s defense is depicted in Figure 2. It shows that in the normal runtime A, an overflow vulnerability can be exploited to jump to an arbitrary address within glibc, but in runtime B with BlankIt, since the library functions are wiped out, jumping to an arbitrary address results in a fault. The prediction mechanism reduces the set of functions that can be reached through the vulnerability.

Figure 2. An application running (A) without BlankIt and (B) with it. The example is based on the return-to-libc attack from [61].

A highly accurate yet lightweight prediction framework is critical to BlankIt’s success. We compute the predicates on

which the call site and call arguments are control-dependent (using reverse dominant frontier), and generate the model based on these predicates and argument values. We empirically show (based on the intuition of static value separation via SSA) that a model based on reverse dominance frontiers and argument values is accurate and lightweight. This approach provides a highly likely subset of the may-reachable set of functions from a given call site, which itself is a function of arguments of the call site summarized by the static-dynamic artifacts in the application. A lightweight decision tree based on these artifacts allows us to predict the may-reachable set of functions at each call site. More details on this are in Section 4. Both under- and over-predictions can occur. An under-prediction is when a function is invoked that was not loaded. It *may* be an attack, so it raises an alarm. An over-prediction is when a function is predicted and loaded but not invoked (which could lead to reduction in security since it is an over-approximation). Over-prediction is very rare (as shown later in Section 5) and is much better than static approximation of reachable functions, as only valid callees are loaded. When under-prediction occurs (which is also rare due to high prediction accuracy), BlankIt starts an alarm and audit phase to check for an attack. The library function and its arguments are handed over to a process that runs the library function with a memory safety mechanism, which then checks if the misprediction is an attack.

3 BlankIt’s Runtime

BlankIt’s runtime wipes out all functions in a library and loads them only on demand. The predictor that guides this Pin-based runtime is described in Section 4.

3.1 BlankIt Design

BlankIt leverages Intel Pin [46], and its design adheres to Pin’s probe mode and programming model. Probe mode allows for fast, native execution (as opposed to Pin’s JIT mode). At initialization time, BlankIt iterates over all of the executable’s shared objects. Then for each function within a shared object, it overwrites the first few bytes with a trampoline and blanks the remaining bytes with illegal or invalid instructions. BlankIt maintains a separate copy of functions in a trusted cache. At runtime, trampolines bounce execution into a generic handler, or in Pin-speak, a probe. The BlankIt probe copies the function’s bytes back into place (i.e. from the end of the trampoline to the end of the function) and returns to the function.

Figure 3 depicts this mechanism for a two-function call chain. When the application calls a library function (`malloc` in this case), execution proceeds along arrow 1 to the original function location in libc. Note that during initialization, however, BlankIt had replaced it with a trampoline. The trampoline causes execution to flow along arrow 2 to the `probe_copy` function. Focusing first on lines 6-9, a loop

over a map m copies and remembers all functions of the currently predicted set. `malloc` and `mmap64` are needed, so they are copied back into place in `libc`. After returning from the `probe_copy` function along arrow 3, execution flows along arrows 4, 5, 6, and 7 when `malloc` invokes `mmap64`.

Figure 3. An example of how the copy probe works. Execution flows in order along the arrows.

Notice that the trampoline in `mmap64` is still taken (the two small dashed arrows at 5 and 6), even though its code is already in place and no additional copying is needed. This is handled in lines 4-5 in `probe_copy`. There is no blanking at the end, either. That is, arrow 8 proceeds back to the application. The copied functions are blanked during the prediction of the next library function call chain, but only if the predicted set has changed. This *lazy blanking* optimization trades some security (by allowing the last call chain in the library to remain unblanked and exposed until the next library call) for speed. It still enforces security across call sites, though, since no two call chains remain linked to the application at the same time.

A second type of probe (Figure 4) is used for both blanking and prediction. In a BlankIt-enabled application, prediction calls are hoisted before the library calls. The predictions are *themselves* library calls (to `libBlankIt.so`), though, and BlankIt simply replaces them with a probe during initialization. In Figure 4, the application calls a library function (`free`), so at compile-time, a call to `blankit_predict` was inserted with an ID of the profiled/learned call chain at that call site. The initialization for BlankIt replaced all `blankit_predict` calls with `probe_blankit_predict`, which allows it to blank the last set of copied functions (lines 4-5) (lazy blanking) and update the ID of the new prediction set (line 6). `probe_blankit_predict` then returns along arrow 2 to the application, where it invokes the library function, `free`. At this point, the behavior is similar to that described in Figure 3. Any misprediction raises an alarm and invokes an audit mechanism.

3.2 Handling Misprediction

The call flow deviates from the predicted path either because of an attack or misprediction. BlankIt’s runtime must distinguish between mispredictions and attacks, and there are

Figure 4. A prediction call that passes an ID of 4 for the upcoming predicted call chain. Within the prediction probe, the last chain of exposed functions will be blanked if the predicted set has changed. In this case, it has changed (from `malloc` and `mmap64` to `free`), so blanking occurs.

several existing approaches that can be plugged into BlankIt with zero to few changes.

Different misprediction handling approaches have different security-performance tradeoffs. The right approach for handling the alarms depends on the application and the level of security required. If the application is 100% predictable, then even killing the process is an option. In our experiments, we show our prediction is 100% accurate in a few applications. Another option is to use a virtual machine (VM) with checkpointing, and on mispredict, restore the application with heightened monitoring [71]. Yet another option is to feed the application state into a memory forensics tool.

Memory forensics is an active area of research with a number of tools that could serve as a drop-in solution for handling BlankIt alarms. One recent tool in particular, CREDAL [69], is a forensics tool that, given a core dump, can identify a data dependency mismatch (indicating an attack or corruption) with 80% success rate. 6% of the failure cases can be resolved by unwinding the stack as required by CREDAL. Further improvements in CREDAL-like systems will reduce the gap in the remaining 14% failures.

In this work, we illustrate one more option: an audit mechanism that checks for memory safety [38]. BlankIt enters an audit and alarm phase on mispredict, in which the library function arguments and the function name are passed to an auditor process. The auditor loads the library with `dlopen` and calls the library function while executing with a memory safety mechanism. Here we consider Valgrind’s Memcheck [39, 53]. Memcheck detects only pointer targets that fall outside of valid allocations, but an experimental extension, SGCheck, can be enabled with `-tool=exp-sgcheck` to handle global and stack overruns. So while Memcheck is not complete, we chose to demonstrate BlankIt with it because it is an easily available off-the-shelf tool for verifying the feasibility of a memory safety mechanism. It also suffers

from high overhead ($\sim 20\times$), making it a useful data point for a “heavy-weight” solution.

We outline the three cases that have to be handled for Memcheck integration to work properly: library calls that have scalar arguments, that have pointer arguments, or that maintain state. Scalar arguments are the simplest. At runtime and on mispredict, the arguments are passed directly to the auditor. Pointer arguments require the auditor to track metadata of the pointer’s start address, type, and size. These can be captured from calls such as `malloc`. An example is passing metadata on the `src` and `dest` pointers in a call to `memcpy`. Library calls that maintain state cannot be independently repeated by Memcheck with just the input arguments. These calls have to be identified prior to runtime, and Memcheck must repeat all calls needed to recreate the state. In order to reduce work, stateful calls only need to be repeated up to the last “reconciliation point” (i.e. wherever the last mispredict occurred and the global state was last synchronized within the auditor).

4 Call Graph Prediction

BlankIt runtime is driven by the prediction component, which is concerned with determining the expected flow through the libraries. Distinguishing between input-specific, legitimate dynamic control-flow and an illegitimate counterpart is critical to thwarting attempts to construct a malicious security apparatus inside the library. The goal is to build an oracle that takes into account the input relatedness and predicts the functions that correspond to the valid control flow.

4.1 Overall Prediction Framework

1. The static divergence analysis finds statically non-divergent functions for which no prediction is needed.
2. We instrument the reverse dominance frontiers (RDFs) corresponding to every argument at every call site to construct a call context for the library call.
3. The instrumented application is profiled, and the call context and argument values are logged along with the call chain.
4. This profile information is fed into a machine learning model that constructs a decision tree to predict the call chain depending on the call context and argument values.
5. The generated decision tree is then instrumented into the application at the application call site for prediction.
6. Constant folding and dead code elimination is carried out to remove redundant predicates.
7. The instrumented application feeds the predicted call chain into the BlankIt runtime system.

4.2 Static Call Flow Divergence Detection

In the first step of the prediction framework, we find library functions that do not have statically divergent call flow. If a library function is non-divergent, all dynamic calls of the function will result in the same call chain, making prediction trivial. We find non-divergent functions by checking if every call site within a function post-dominates the entry block of the function, as shown in Algorithm 1. If the callee is not on a control-divergent path but is itself divergent, then the caller is also marked as a divergent function, and all other functions that call this function are also marked as divergent functions.

Algorithm 1 Static Divergence Detection

```

1: procedure NON-DIVERGENT(Function F)
2:    $BB_e \leftarrow$  entry basic block in F
3:   for each Call site  $C_i$ :
4:      $f_{ci} \leftarrow$  function called in  $C_i$ 
5:     if NON-DIVERGENT( $f_{ci}$ ) &&  $C_i$  post-dominates  $BB_e$ 
6:       then
7:         F is non-divergent
8:   End for
```

In glibc, we found that among all the functions analyzed, 737 functions (roughly 27%) are statically non-divergent. For the remaining 1985 functions (73%), dynamic prediction is a must.

4.3 Reverse Dominance Frontier-Based Prediction

After generating a list of statically non-divergent functions, we now look at predicting the call graph for the diverging ones. Normally, whole program path profiling [4] would need instrumentation of every branch. This would result in a path prefix of a fixed window W , entailing the context of the last W branches. This path prefix can then be used for prediction of the call chain within the function call, but this elaborate model could also pose very high overheads. We leverage the fact that library functions are usually control-divergent due to their arguments and contend that the static context required to predict the call chain is dependent upon the control dependence of the arguments passed to the library.

This motivation leads us to investigate the reverse dominance frontier (RDF) of the arguments of a call site, because the divergence in the library call path can be caused by the different definitions of arguments (determined by their respective RDF) reaching the call site. For example in Listing 1, the input to function `service` in `access_service` has one argument key, which is defined at basic blocks $B1$ and $B2$. If $B1$ is executed, then key has admin access, and if $B2$ is executed, it does not. In this case, both $B1$ and $B2$ are control dependent on the branch `if(auth)` at Line 14. Thus the branch condition decides the functions invoked within

service. The intuition here is that the call chain inside a library call will be dependent on arguments sent to the library, which in turn could be statically separated into corresponding SSA variables of a backward slice that are finally guarded by the control-dependent branches. With RDF-based prediction, the context leading up to a call site consists of only the last executed RDF for every function argument. This minimal instrumentation has enough information to predict the call chain within a function with reasonable accuracy. When the profiling is done, the context for each call site is constructed separately, depending on the RDFs of its arguments. The instrumentation methodology is outlined in Algorithm 2.

Algorithm 2 RDF-based Instrumentation

```

1: procedure INSTRUMENT_FUNCTION(Func)
2:   Bbs_to_instrument  $\leftarrow$  Null
3:   for all callsite(Ci)  $\in$  Func do
4:     Fci  $\leftarrow$  function called in Ci
5:     for all argument Ai  $\in$  Fci do
6:       def_worklist  $\leftarrow$  get definition of Ai
7:       for all instruction defi  $\in$  worklist do
8:         phi_instr_set  $\leftarrow$  TRACE_PARENT_PHI(defi)
9:          $\triangleright$  call returns a set of phi instructions
10:      for all  $\phi_i \in$  phi_instr_set do
11:        for all incoming_BBi  $\in$   $\phi_i$  do
12:          rdf_set  $\leftarrow$  GET_RDF(incoming_BBi)
13:        for all rdf_BBi  $\in$  rdf_set do
14:          for all Successor Succi  $\in$  rdf_BBi do
15:            Bbs_to_instrument  $\leftarrow$  Succi
16:      return Bbs_to_instrument
17: procedure TRACE_PARENT_PHI(instruction)
18:   phi_instr_set  $\leftarrow$  Null
19:   for all Operands opi  $\in$  instruction do
20:     if opi is a phi instruction then
21:       phi_instr_set  $\leftarrow$  opi
22:     else
23:       phi_instr_set  $\leftarrow$  TRACE_PARENT_PHI(opi)
24:   return phi_instr_set

```

4.4 Argument Value-Based Prediction

The control path in library functions can diverge because of the various values of the arguments that are reaching the call sites in the application from the same RDF. Since the RDF remains constant, RDF-based prediction might not be sufficient. For example, we have observed that math library functions like $\sqrt{x}$ and $\exp(x)$ have different call chains depending on the range of the parameters. In order to handle such cases, we must classify the values of the function arguments that reach from a single RDF, so we also include the values of the arguments captured by profiling in training the model. The value-based profiling also captures the values of

the function pointers. Thus if a function pointer is passed as an argument to another function, then we are able to predict the function invocation based on the function pointer values.

4.5 Decision Trees

To predict the library call chain at each call site using RDF and call arguments, we chose decision trees as our machine learning model. Decision trees can be easily embedded into the application as few predicates without increasing the application runtime overhead. First, using LLVM, we instrumented the calls for recording the call context for every library call site within the application. The instrumented application was run using Pin JIT to generate the profile trace (the library callgraph), which along with the call context is then parsed to create the training data. The output of our model is the list of functions called within each library. The training data consists of the following features:

- Application call site
- Top-level library function called from application (callee name)
- Reverse dominance frontier of arguments
- Arguments passed to the library call.

For example in Listing 1, the decision tree learns that on a call to service from function access_service, call_super_user is called only when RDF B1 is taken or *key* < 750, as shown in Figure 5. We leverage the python-based scikit library [48] (sklearn.tree.DecisionTreeClassifier API), to learn a decision tree over the training data. The decision tree model is then written to a file in a special text format, which is read by an LLVM pass that embeds the decision tree within the application for each library call. The decision tree is basically a binary tree of predicates of input features. The tree follows a certain path at each predicate based on the predicates' outcome, and halts when it reaches a leaf node. The leaf's "class label" is the set of functions within the library predicted to be called at the call site. We experimentally found that a max tree-depth of 10 provided good accuracy with acceptable performance. Higher depth would only be required when there are too many rows of the same input features but different outcomes, though this would also add the dilemma of overfitting the training data. We did not use boosting for learning the decision tree.

5 Evaluation 1: SPEC CPU 2006

We evaluate BlankIt using SPEC CPU 2006, followed by two real-world applications in the subsequent section.

SPEC has several properties that make it a good candidate for exercising and evaluating BlankIt. First, it is performance-oriented, so we can measure runtime overheads on realistic CPU-intensive benchmarks. Second, it has a standard library profile, which carries a relatively large CVE list. Lastly, callgraphs of SPEC benchmarks and their supporting libraries

Figure 5. Example decision tree for Listing 1

are significantly divergent, which is needed to stress the prediction framework (see Table 1).

Table 1. KEY GRAPH METRICS FOR SPEC CPU 2006 AND GLIBC

SPEC benchmark and glibc graph metrics	Avg
1. # dynamically called glibc funcs from SPEC	63
2. # statically reachable funcs in glibc based on (1)	171
3. # callgraph edges based on static callgraph in (2)	378
4. # non-divergent functions in (2)	8
5. max static callchain depth of (2)	7

With the exception of the linker shared object and the vdso image, we have instrumented all the benchmarks’ shared objects. The most important is glibc-2.26 [23], a fundamental library for programs compiled on GNU/Linux systems. The library provides POSIX, BSD, OS-specific and other APIs including utilities like print, login, and crypt. Historically, it has also been targeted and exploited heavily. The other libraries are libm 2.26, libgcc_s from the libgcc1 package v1:7.2.0-1ubuntu1 16.04, and libstdc++ 6.0.24, which are all tackled by BlankIt. Our runtime uses Pin v3.6. We have built SPEC with version 5 of LLVM, which fails to compile 400.perl-bench. It generates multiple definitions for gnu_dev_major, minor, and other functions, so we have elided treatment of it here. All other C/C++ benchmarks are presented. The raw decision trees have multiple compile-time constant checks. Constant folding reduces these trees to relatively few *if-else* checks, increasing the binary sizes by at most 1%. All runtime overhead experiments are on an Ubuntu 16.04.3 LTS machine with an AMD Ryzen 7 1800X 8-Core 3.6 GHz processor and 32 GB DDR4 2666 MHz RAM. All the SPEC benchmarks were trained on “test” (small) and “train” (medium) input data sets and were tested using the (large) “ref” input data sets. All runtime experiments are averaged over three runs.

5.1 Security

What constitutes an attack surface, especially when preparing for unknown future attacks, is difficult to capture. We present three results for reducing dynamically linked functions. Taken together, they offer three different axes for understanding how BlankIt is behaving. They summarize (1) the number of functions that are exposed, (2) the number of gadgets in a known type of attack that are exposed, and (3) the number of known functions from a vulnerable list that are exposed. As additional evidence, we also provide a measure of gadget expressiveness, i.e. how easy-to-use or useful a gadget is to an attacker. We show how BlankIt not only dramatically reduces the number of gadgets available to an attacker but also makes the available gadgets more difficult to use for an attacker.

The first measure is a dynamic metric that describes the maximum number of library functions exposed at any given time during execution. This can be described by the following formula:

$$exposed = p + s + c \quad (1)$$

where *exposed* is the maximum number of loaded functions at runtime; *p* is the number of functions that Pin is unable to instrument and thus must be left as they are (without blanking them); *s* is the number of functions that are less than 14 bytes and therefore too small for blanking; and *c* is the maximum-length call chain for a given benchmark that our framework dynamically loads at any call site during execution. In other words, the number of exposed functions in the worst case is the maximum number of unblanked functions that could be leveraged in some attack. The percent reduction of code surface is then given by:

$$reduction = \frac{\sum_l n_l - exposed}{\sum_l n_l} * 100 \quad (2)$$

where *l* is some library and *n_l* is the total number of functions available at runtime in some library *l*. This metric does not capture anything about the inherent weaknesses or strengths of the exposed functions - only that they are exposed. Table 2 presents this result in the first column. The worst-case reduction is, on average, 97.1%, which is significantly higher than any static or dynamic cutting/debloating technique. For example, worst-case function reduction in piece-wise [49] is 60% and 46.09% on musl-libc and C++ libraries, respectively.

Return-oriented programming depends on ROP gadgets in the code in order to carry out an effective attack. To measure the reduction in gadgets, we leveraged ROPgadget [51], an analysis tool for enumerating the ROP gadgets in a binary. Because BlankIt is a dynamic technique, the number of gadgets varies over time. Thus, similar to before, we choose the worst-case scenario at any given point during the program execution and report its ROP gadget reduction. We calculate the maximum number of exposed gadgets (similar to equation 1). Then we calculate the reduction by summing

Table 2. SECURITY-RELATED IMPROVEMENTS IN SPEC CPU2006

Benchmark	% Exposed Code Surface Reduction	% ROP Gadget Reduction	% glibc CVE Function Reduction
401.bzip2	97.7	98.9	95.7
403.gcc	97.2	97.8	95.7
429.mcf	94.5	95.9	95.7
433.milc	98	98.5	95.7
444.namd	97	96.4	93.6
445.gobmk	95.7	96.2	93.6
450.soplex	97.4	99.3	97.9
453.povray	96.9	97.7	93.6
456.hmmer	97.9	97.9	93.6
458.sjeng	97.8	98.9	95.7
462.libquant	97.9	98.6	95.7
464.h264ref	97.9	98.7	95.7
470.lbm	97.8	98.6	95.7
471.omnetpp	95.6	96.9	95.7
473.astar	96.6	96.6	95.7
482.sphinx3	97.6	98.2	95.7
483.xalanc	96.9	98.1	91.5

over the number of gadgets in the text section in each library, subtracting out the exposed gadgets, and gathering the percentage (as in equation 2). Table 2 shows the benefits of BlankIt on ROP gadget reduction. The average is 97.8%.

The last metric is a measure of the functions in glibc CVEs that are removed by BlankIt. We reviewed the list of all the CVEs for glibc, which reach back to year 2000, and we identified all unique functions mentioned in the descriptions. Of these, we identified 47 that are unequivocally loaded by glibc in the SPEC suite. That is, there must be an exact match in the dynamically loaded list of glibc functions for a function related in a glibc CVE to be considered. For example, “alloca” is mentioned in CVE-2015-1473, but it is not explicitly exported with that name and so is discarded, even though there are multiple allocation functions. Then the number of exposed CVE functions is obtained as follows:

$$exposed_{cve} = p + s + a \quad (3)$$

where p and s are uninstrumented due to Pin or too small to instrument (as before), but a represents *any* function that is called that is in the CVE function list. The percentage is then taken out of the 47 CVE functions. This metric should not be misunderstood as the number of CVEs since 2000 that are thwarted by BlankIt. Rather, it classifies functions as vulnerable based on their CVE history, and then asks how many such functions are exposed under BlankIt running a common benchmark. The results are shown in the last column of Table 2. The average reduction is 95.1%.

We also measure the quality of gadgets with and without BlankIt using Gality [20]. Gality provides an overall score for a notion of the ease with which one could construct an attack. Gality first categorizes the gadgets (arithmetic, data move, etc.), finding the distribution of different gadgets that an attacker could use in their arsenal, and then checks for side effects of a gadget. A side effect is measured by the number of preconditions that must be satisfied for using a gadget. In other words, a high-quality gadget is one with no preconditions or that does not affect any other register or memory. The Gality score starts at 0 and is increased for side effects and preconditions. Thus, a higher score indicates worse gadget quality and hence better security. The authors of Gality use the value of 1 as a cutoff and indicate that gadget quality below 1 is of “high quality” to an attacker. To summarize, the BlankIt-enabled SPEC applications have higher scores in all cases (with an average of 1.09, versus 0.77 without BlankIt), signifying improved security. In 15 out of 17 cases, scores are near or above 1, implying high security.

The metrics in the community for assessing the gadget quality are still limited, a concern that has been expressed in previous CFI works (e.g. [10]), and other metrics have been proposed (e.g. the AIR metric in [72] for indirect jump reduction). [8] recently attempts to carefully quantify gadget expressiveness. Although only one gadget can be sufficient to hijack a system, our gadget reduction and quality results are nevertheless useful for comparison with other similar debloating works such as [25, 49, 50, 55].

Lastly, note that multiple applications have their own private copy of shared libraries at runtime. Thus, even if multiple BlankIt-enabled applications run simultaneously and unblank disjoint library function sets, process isolation prevents any “union of exposure” among them.

5.2 Performance

Figure 6 shows the runtime overhead. The most crucial library for security purposes is glibc. The average slowdown for a BlankIt-enabled application on only glibc is 16%. There is a slight speedup in the case of lbm attributed to instrumentation effects on memory alignment as in other security works [27, 42, 72]. The worst case is gcc, with a 1.76x overhead. Note that runtime overhead is not a function of misprediction in the current implementation, as alarms are handled in parallel. Thus, while prediction accuracy was poorest for libquantum (discussed in the next section), the performance was good. Covering all libraries costs little in overhead for the extra security. The average slowdown for BlankIt over all libs is 18%, demonstrating that BlankIt likely scales well across programs’ dynamic library sets. (Note that in bzip2, gcc, mcf, and sjeng, libc is the only dynamically linked library.)

The runtime overhead for BlankIt depends on several factors. We see that BlankIt invokes (and handles) a non-trivial

number of dynamic library calls for SPEC: All but four benchmarks invoke over 1M calls, and another four make over 100M calls. The timing is more complex than just the dynamic call counts, though. For instance, *sjeng* makes 24,766 dynamic library calls, but its performance is 1.5x worse than the native baseline; compare this with *povray*, with 4,323,690,635 dynamic calls and 1.1x slowdown. The overhead depends on the ratio of the program's runtime spent in libraries, and it also depends on the ratio of the time to copy a function (which depends on its size and other factors like alignment) versus the time to execute that function. Optimizations like lazy blanking can help, by leaving functions unblanked when they are called successively. We looked at the effect of the lazy blanking optimization, and *hmmer* suffered the most without it (4.61x slowdown when disabling lazy blanking); *gcc* saw the least benefit from it (1.05x slowdown when disabling it).

We looked at the latency of the BlankIt probes to see where it was adding time. The trampolines and probe logic are low costs ($<1\mu s$), but un/blanking dominates due to *memset/cpy* (up to 10s of microseconds without lazy blanking). For example, in *gcc*, the average runtime cost of a full blanking probe is roughly 5x compared to a lazy one (i.e. a trampoline + *memset* vs. just the trampoline). Similarly, a full unblanking probe versus its lazy counterpart costs over 10x. For perspective, the absolute time, on average, to unblank code was $2.57\mu s$ in our *gcc* experiments (max of $37\mu s$), versus $0.22\mu s$ in lazy cases. We considered a simple dynamic runtime scheme that would keep in memory only the union of functions seen during profiling (on a call-site basis). This increased loaded library functions in SPEC by 58%. This hurts security, but the increased footprint also hurts performance. Because un/blanking costs dominate, prediction is necessary to reduce the memory copying and setting.

5.3 Prediction

Our treatment of prediction includes our observations while working with the framework, a breakdown of the mispredictions, and our accuracy results and analysis on SPEC.

5.3.1 Observations. In the case of library functions that usually do not have an argument context for library function calls, we have observed that the set of functions called within the library can be predicted based on the dynamic calling context. The necessary dynamic calling context is discovered by the decision tree, which can be as simple as just the name of the function and/or call site. However, sometimes the call chain is dependent on the arguments passed to the function, which could include a function pointer. For example, there might be a math library function that calls a different set of internal functions based on the arguments. The decision tree is able to handle such cases by training on the values passed to the function. It is able to model the different call chains that are invoked based on the range of values of

the function parameters encountered during profiling. The decision tree is currently unable to capture other use cases like certain special system states or error conditions that were not triggered during profiling, which is a challenge of any machine learning approach.

5.3.2 Breakdown of Misprediction in SPEC. We classify the mispredictions into over-prediction (where more functions were loaded than necessary) and under-prediction (where fewer functions were loaded than necessary). In all but two cases, under-predictions accounted for 100% of mispredictions. The exceptions are *gcc*, where the under- and over-predictions were almost equal (0.5% difference), and *soplex*, where under-predictions accounted for 93% of the cases. The main reason is that when the decision tree encounters a call site which was not exercised during training, it is extremely conservative and bases the prediction on whether the entry function was seen during training, and predicts a callee set which is a subset of the static callee set for an entry function. This in turn provides good security, since no spurious functions are brought in that could increase the exposed code surface.

5.3.3 Accuracy and Audit Overhead. Table 3 shows the call chain prediction accuracy and audit overhead for the SPEC2006 benchmark suite. The audit overheads are based on the expected runtime of *parallel* Valgrind threads on each mispredict. In summary, only two benchmarks would require more than 1 parallel Valgrind process to keep pace with mispredicts. Over half of the accuracy figures are 97% or greater; three are 100%; and the average is 94.3%. In fact, the decision tree's prediction ability was tested for predicting every function in SPEC2006 at every call site and was found to be very close to the one reported here for the libraries.

One important reason for very high prediction accuracy stems from the fact that libraries follow software design patterns; the use cases of libraries are therefore finite and can be siloed. Our decision trees are able to learn these input-related patterns and during testing pinpoint the silo and its underlying set of function calls. The only application for which the decision tree accuracy was relatively low is *libquantum*, in which a call chain involving function *cfree* in training is followed by *munmap* during testing because of the size of the memory that is freed in *cfree* in the train vs. ref input.

6 Evaluation 2: nginx and sshd

We evaluate two real world applications: *nginx* v1.15.10 [40], a common server and reverse proxy tool, and *sshd* v8.0p1 [45], the server daemon for the *openssh* communication and encryption tool suite. Both of these applications serve as critical infrastructure in today's software landscape, and both of them suffer from real-world attacks that exploit vulnerabilities in linked libraries. We evaluate them on security,

Figure 6. Runtime slowdown for BlankIt on SPEC 2006 CPU, normalized against native.

Table 3. PREDICTION ACCURACY AND AUDIT OVERHEAD

Benchmark	% Accuracy	Audit (μ s)
401.bzip2	91	287
403.gcc	99	12
429.mcf	94	92
433.milc	100	27
444.namd	99	11
445.gobmk	84	8
450.soplex	92	26
453.povray	97	6
456.hmmer	98	6
458.sjeng	97	27
462.libquantum	60	41
464.h264ref	100	28
470.lbm	98	171
471.omnetpp	99	6
473.astar	100	58
482.sphinx3	99	11
483.xalancbmk	97	18

performance, and prediction, and study BlankIt’s defense against well-known attacks. Note that nginx is both a multi-processing and multithreaded application. In a multithreaded environment, un/blinking can be implemented with `mremap` without marking the page writable. This protects against a nefarious thread attempting to write to a page that is being unblanked.

To train BlankIt’s prediction mechanism for nginx, we used a basic test page (only html and body tags that reads “site under construction”), and made requests for 3s. To perform the requests, we used a traffic generation tool called wrk [64] (used by Nginx, Inc. for benchmarking). We ran

wrk using 12 threads with 400 connections. To train sshd, we used a simple ssh payload script which echos “hello world” on the server and sends it back to the ssh client; we also spawned 1000 such concurrent connections in order to train with sshd’s error handling.

6.1 Security

Our security results for nginx are based on running a 30s test with a standard 80KB file, the main page for Wikipedia [63]. In the case of sshd, we use a shell script that implements Dijkstra’s algorithm to solve a shortest path graph problem that executes for approximately 40s. Table 4 shows security results for nginx and sshd similar to those achieved in SPEC. Exposed code surface reduction is 95.8% and 89.5% for nginx and sshd, respectively. Less than 2% of the ROP gadgets remain exposed. Gality gadget quality under BlankIt is again greater than 1 (while the unprotected version is ~25% worse). The number of gadgets in the nginx and sshd libraries are 27,523 and 30,752, respectively, and the maximum number of exposed gadgets under BlankIt are merely 299 and 562. These two applications also offer a look at the effectiveness of BlankIt on additional libraries. Linked libraries in nginx were libc, libcrypto, libdl, libpcre, libpthread, and libz; in sshd, they were libc, libcrypto, libdl, libutil, libz, libcrypt, libresolv, libssh, libnss_compat, libnsl, libnss_nis, and libnss_files. We ignored only the vdso image and the linker shared object.

6.2 Performance

For nginx, we ran three performance experiments. Table 5 shows the effect of running nginx under a constant load for varying amounts of time, verifying that it is not degrading. BlankIt overhead is in the range of 1.06x to 1.09x. We used the Wikipedia main page for this experiment. Table 6 shows

Table 4. SECURITY-RELATED IMPROVEMENTS FOR NGINX AND SSHD

Benchmark	% Exposed Code Surface Reduction	% ROP Gadget Reduction	% glibc CVE Function Reduction
nginx	95.8	98.9	93.6
sshd	89.5	98.2	91.5

the result of running BlankIt under the same 12-thread, 400-connection workload, but for 30s under varying payloads. Here we used random, binary payloads of size 1MB, 10MB, and 100MB. We see that running nginx with BlankIt scales with the sizes of the requests. We checked the CPU utilization and instructions per cycle (IPC) with perf for these experiments. We found no noticeable difference in CPU utilization. In the former experiment (varying length of sustained time), the IPC degraded by 3.3% on average. In the latter experiment (varying the payload size), the IPC degraded by 0.9% on average. There appears to be no mounting overhead for running nginx with BlankIt for an extended time, but shorter payloads (e.g. the size of a single HTML page) affect the IPC slightly more. We ran one additional performance experiment on nginx: a glibc-only protected version. We ran the same wrk configuration as before, for 30s, using the main page for Wikipedia. The slowdown of bytes transferred per second was 1.07x, which is comparable to the overhead when supporting all libraries.

To test sshd's performance, we ran the Dijkstra shell script against vanilla and BlankIt-enabled sshd. We measured no noticeable runtime difference. For this experiment we also checked CPU utilization and IPC. We found a negligible difference in CPU utilization (1% worse with BlankIt) and detected no difference in IPC. We tested with concurrent ssh connections and again found no noticeable effect. The reason appears to be that the interaction with ssh clients are completed immediately and the control passed to spawned shells. We tried with more connections (up to 1000) and still saw no difference. Thus, for openssh, the security benefit of BlankIt comes with little to no performance penalty.

Table 5. TRANSFER/SEC DEGRADATION WITH BLANKIT UNDER VARYING, SUSTAINED LOAD TIMES.

Benchmark	3s	30s	300s
nginx	1.07x	1.09x	1.06x

Table 6. TRANSFER/SEC DEGRADATION WITH BLANKIT UNDER VARYING, SUSTAINED REQUEST SIZES.

Benchmark	1MB	10MB	100MB
nginx	1.02x	1.04x	1.00x

6.3 Prediction

For nginx, we tested against 2 kinds of requests: a normal-sized html page (the main page for Wikipedia) and a randomized binary file (1MB). In both cases we ran for 30s. The results are shown as part of Table 7. BlankIt's prediction for nginx is 100% in both cases. In other words, the call chains within dynamic libraries for nginx are easily predictable under normal execution for GET requests, and this is predictable even with minimal training.

To test prediction on sshd, we designed several scripts. The first manipulates a few folders, changes directories, and writes "hello world" strings to files and reads them back. In Table 7, this is the "cmds" test. The "cmds-100" test is the same but with 100 concurrent connections. The Dijkstra test is the same script used in performance and security testing. We performed a 100 concurrent-connections test for it, as well. Lastly, there is a "6MB" test that simply reads from a 6MB text file (retrieved from Peter Norvig's website [44]) and sends it back to the client through the terminal. Our results show that with minimal training, BlankIt predicts with 95% accuracy and above for these typical ssh interactions.

There is no measurable audit overhead for nginx because the accuracy was perfect in our experiments. This is with non-trivial dynamic library call counts (11,521,794 and 2,547,380 library calls in 30s for the Wikipedia and 1mb.bin tests, respectively). In sshd, the call counts are lower (ranging from roughly 2,000 to 50,000, depending on the input), and the highest mispredict / second ratio is under 300. Table 7 shows audited functions take up to $\sim 100\mu s$ to execute. Thus with 300 mispredicts / second, an alarm handling mechanism only needs about 30ms of that 1 second to process alarms, which is more than enough to keep pace in parallel.

Table 7. CALL CHAIN PREDICTION ACCURACY AND AUDIT OVERHEAD FOR NGINX AND SSHD

Benchmark	Test	% Accuracy	Audit (μs)
nginx	wiki	100	36
nginx	1MB	100	40
sshd	cmds	97	53
sshd	cmds-100	95	47
sshd	dijkstra	97	56
sshd	dijkstra-100	98	73
sshd	6MB	97	55

6.4 Defense Against Real-world Attacks

We describe two known attacks on these applications and show how BlankIt defends against them: the attack demonstrated in "Control Jujutsu" [18] against nginx, and CVE-2015-7547 for sshd [56].

The Control Jujutsu [18] attack on nginx proceeds as follows: A memory vulnerability is exploited, allowing an attacker to modify a function pointer and its arguments; when

the function pointer is eventually invoked, execution jumps to the attacker’s target function with the corrupted arguments. In the paper, an nginx global object called `ctx` is corrupted, modifying the `ctx`’s `output_filter()` function to target `ngx_execute_proc()`, which executes `execve()`. The arguments, which are passed via `ctx`’s `filter_ctx` member, are also corrupted. This allows for the attackers to carry out remote code execution. The authors show this is all achievable using just HTTP GET requests. As we observed, BlankIt’s prediction on nginx is very strong (100% in our GET experiments using simple training), which suggests any kind of function pointer manipulation will be caught by BlankIt.

The `sshd` vulnerability allows “remote attackers to cause a denial of service (crash) or possibly execute arbitrary code via a crafted ... call to the `getaddrinfo` function [in `glibc`].” As we showed in our evaluation, BlankIt can be enabled for `sshd` to protect `libnss` modules and `libresolv`, which were involved in the attack. In short, the attack can be triggered by `ssh`, and it is based off a mistake in `glibc` that uses an undersized stack buffer when it should be using its heap; this leads to an overflow and potentially arbitrary code execution. With BlankIt a jump to an arbitrary location for using library gadgets would enter a blanked region, and the unpredicted function execution would raise an alarm.

7 Related Work

The two main lines of research that are closely related to our work are security-based software debloating and security techniques that mitigate memory corruption.

In the Introduction, we contrasted BlankIt with sound debloating techniques like static debloating and load-time mechanisms like piece-wise [49], and with unsound techniques like Razor [50] and Chisel [24]. BlankIt differs in goals and achievements, and direct comparisons are challenging. To summarize, however, BlankIt’s ROP gadget reduction is 97% or greater on SPEC with `glibc`, with 18% runtime overhead (all libs). To our knowledge, no other security debloating mechanisms handle `glibc`. Piece-wise uses `musl-libc` (2 CVEs to `glibc`’s 108) and achieves up to 86% ROP gadget reduction on SPEC. Chisel and Razor are debloating only applications; they achieve 85.1% and 68.19% code reduction on SPEC; they are unsound.

Within the software engineering community several works [32–34, 65–68] have targeted debloating to improve performance. The embedded software community performed debloating based on link-time optimizations to reduce code size. The key goal of compiler-based link-time optimizations is to compact the code size [5, 15] [21, 37]. While these works have paved the way for debloating, they are not geared towards solving the security issues due to bloated libraries.

We also discussed the different CFI and memory safety mechanisms that avoid change in control-flow due to memory corruption and prevent memory corruption through

memory safety, respectively. While BlankIt is not a stand-alone system and depends on CFI for thwarting attacks, BlankIt strengthens CFI by reducing the code surface and disallowing the use of gadgets in the library code for attacks that bypass CFI. BlankIt also catches attacks such as non-control data attacks that involve change in call flow by running an audit mechanism with memory safety on mis-predictions. An online full memory safety mechanism can incur overheads greater than 100%. However, full memory safety mechanisms like AddressSan [52] or EffectiveSan [16] can be employed offline to detect bugs in the code and are used for a different purpose.

8 Conclusion

In this work, we propose and demonstrate a “get what you want” security defense scheme that effectively reduces the exposed code surface of vulnerable linked libraries and defends against attacks. This is a departure from current debloat techniques that “cut what you don’t want.” As a security technique, BlankIt advances the state of the art in binary library debloating and in defenses against non-control data attacks. As a debloating technique within the security space, BlankIt achieves aggressive, sound function and gadget removal. At the crux of our approach is a decision-tree based call path predictor. The predicted call graph is used by a binary-level mechanism that dynamically loads the needed functions at a given call site while the unneeded ones are blanked, which significantly weakens an attacker’s ability to construct gadgets. Our results on the SPEC2006 benchmark suite (evaluated with `glibc`) show prediction accuracy in most cases to be at least 97%, and dynamically linked function and ROP gadget reduction likewise to be at least 97%. On average, the runtime overhead with BlankIt is 18% across all libraries. We also evaluate two critical server applications, `nginx` and `sshd`, with over 10 more libraries, and show similar security and prediction results, with overhead being either within 10% or negligible, respectively. Due to near perfect reduction in the gadgets and CVEs with reasonable overheads, we conclude that BlankIt provides a practical debloating mechanism.

Acknowledgments

The authors gratefully acknowledge that this work was partially supported by the Office of Naval Research (ONR) via Grant No. N00014-17-1-2895, Period: September 1, 2017 to August 31, 2022. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Office of Naval Research.

References

- [1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA*,

- November 7–11, 2005, Vijay Atluri, Catherine A. Meadows, and Ari Juels (Eds.). ACM, 340–353. <https://doi.org/10.1145/1102120.1102165>
- [2] S. Andersen and V. Abella. 2004. Data Execution Prevention: Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. <http://technet.microsoft.com/en-us/library/bb457155.aspx>. Accessed: 2018 May 7.
 - [3] ANDK. 2017. Android NDK. <https://developer.android.com/ndk/guides/cpp-support.html>. Accessed: 2017 Feb 5.
 - [4] Thomas Ball and James R. Larus. 1996. Efficient Path Profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 29)*. IEEE Computer Society, USA, 46–57.
 - [5] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. 2003. Survey of Code-size Reduction Methods. *ACM Comput. Surv.* 35, 3 (Sept. 2003), 223–267. <https://doi.org/10.1145/937503.937504>
 - [6] Tyler K. Bletsch, Xuxian Jiang, and Vincent W. Freeh. 2011. Mitigating code-reuse attacks with control-flow locking. In *Twenty-Seventh Annual Computer Security Applications Conference, ACSAC 2011, Orlando, FL, USA, 5–9 December 2011*, Robert H’obbes’ Zakon, John P. McDermott, and Michael E. Locasto (Eds.). ACM, 353–362. <https://doi.org/10.1145/2076732.2076783>
 - [7] Tyler K. Bletsch, Xuxian Jiang, Vincent W. Freeh, and Zhenkai Liang. 2011. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, ASIACCS 2011, Hong Kong, China, March 22–24, 2011*, Bruce S. N. Cheung, Lucas Chi Kwong Hui, Ravi S. Sandhu, and Duncan S. Wong (Eds.). ACM, 30–40. <https://doi.org/10.1145/1966913.1966919>
 - [8] Michael D. Brown and Santosh Pande. 2019. Is Less Really More? Towards Better Metrics for Measuring Security Improvements Realized Through Software Debloating. In *12th USENIX Workshop on Cyber Security Experimentation and Test, CSET 2019, Santa Clara, CA, USA, August 12, 2019*, Rob Jansen and Peter A. H. Peterson (Eds.). USENIX Association. <https://www.usenix.org/conference/cset19/presentation/brown>
 - [9] Caffe. 2017. Caffe. <http://caffe.berkeleyvision.org/>. Accessed: 2017 Feb 5.
 - [10] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In *Proceedings of the 24th USENIX Conference on Security Symposium (SEC’15)*. USENIX Association, Berkeley, CA, USA, 161–176. <http://dl.acm.org/citation.cfm?id=2831143.2831154>
 - [11] Shuo Chen, Jun Xu, and Emre Can Sezer. 2005. Non-Control-Data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium, Baltimore, MD, USA, July 31 – August 5, 2005*, Patrick D. McDaniel (Ed.). USENIX Association. <https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats>
 - [12] John Criswell, Nathan Dautenhahn, and Vikram S. Adve. 2014. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18–21, 2014*. IEEE Computer Society, 292–307. <https://doi.org/10.1109/SP.2014.26>
 - [13] CVE. 2017. CVE Details. https://www.cvedetails.com/vulnerability-list/vendor_id-72/product_id-767/GNU-Glibc.html. Accessed: 2017 Feb 5.
 - [14] Saumya Debray and William Evans. 2002. Profile-guided Code Compression. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI ’02)*. ACM, New York, NY, USA, 95–105. <https://doi.org/10.1145/512529.512542>
 - [15] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler Techniques for Code Compaction. *ACM Trans. Program. Lang. Syst.* 22, 2 (March 2000), 378–415. <https://doi.org/10.1145/349214.349233>
 - [16] Gregory J. Duck and Roland H. C. Yap. 2018. EffectiveSan: Type and Memory Error Detection Using Dynamically Typed C/C++. *SIGPLAN Not.* 53, 4 (June 2018), 181–195. <https://doi.org/10.1145/3296979.3192388>
 - [17] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard E. Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17–21, 2015*. IEEE Computer Society, 781–796. <https://doi.org/10.1109/SP.2015.53>
 - [18] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control Jujutsu: On the Weaknesses of Fine-Grained Control Flow Integrity. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS ’15)*. ACM, New York, NY, USA, 901–913. <https://doi.org/10.1145/2810103.2813646>
 - [19] Flask. 2018. Flask. <http://flask.pocoo.org/>. Accessed: 2018 Apr 24.
 - [20] Andreas Follner, Alexandre Bartel, and Eric Bodden. 2016. Analyzing the Gadgets - Towards a Metric to Measure Gadget Quality. In *Engineering Secure Software and Systems - 8th International Symposium, ESSoS 2016, London, UK, April 6–8, 2016. Proceedings (Lecture Notes in Computer Science)*, Juan Caballero, Eric Bodden, and Elias Athanasopoulos (Eds.). Vol. 9639. Springer, 155–172. https://doi.org/10.1007/978-3-319-30806-7_10
 - [21] Michael Franz and Thomas Kistler. 1997. Slim Binaries. *Commun. ACM* 40, 12 (Dec. 1997), 87–94. <https://doi.org/10.1145/265563.265576>
 - [22] Xinyang Ge, Weidong Cui, and Trent Jaeger. 2017. GRIFIN: Guarding Control Flows Using Intel Processor Trace. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi’an, China, April 8–12, 2017*, Yunji Chen, Olivier Temam, and John Carter (Eds.). ACM, 585–598. <https://doi.org/10.1145/3037697.3037716>
 - [23] glibc. 2017. glibc. <https://www.gnu.org/software/libc/>. Accessed: 2017 Feb 5.
 - [24] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15–19, 2018*. 380–394. <https://doi.org/10.1145/3243734.3243838>
 - [25] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective Program Debloating via Reinforcement Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS ’18)*. ACM, New York, NY, USA, 380–394. <https://doi.org/10.1145/3243734.3243838>
 - [26] Hong Hu, Chenxiong Qian, Carter Yagemann, Simon Pak Ho Chung, William R. Harris, Taesoo Kim, and Wenke Lee. 2018. Enforcing Unique Code Target Property for Control-Flow Integrity. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS ’18)*. ACM, New York, NY, USA, 1470–1486. <https://doi.org/10.1145/3243734.3243797>
 - [27] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI’14)*. USENIX Association, Berkeley, CA, USA, 147–163. <http://dl.acm.org/citation.cfm?id=2685048.2685061>
 - [28] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, and Dawn Song. 2015. Poster : Getting The Point(er): On the Feasibility of Attacks on Code-Pointer Integrity.
 - [29] Vu Le, Chengnian Sun, and Zhendong Su. 2015. Randomized Stress-testing of Link-time Optimizers. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*. ACM, New York, NY, USA, 327–337. <https://doi.org/10.1145/2771783.2771785>
 - [30] Y. Liu, P. Shi, X. Wang, H. Chen, B. Zang, and H. Guan. 2017. Transparent and Efficient CFI Enforcement with Intel Processor Trace. In

- 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA). 529–540. <https://doi.org/10.1109/HPCA.2017.18>
- [31] Pratyusa K. Manadhata and Jeannette M. Wing. 2011. An Attack Surface Metric. *IEEE Trans. Software Eng.* 37, 3 (2011), 371–386.
- [32] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2009. Making Sense of Large Heaps. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings (Lecture Notes in Computer Science)*, Sophia Drossopoulou (Ed.), Vol. 5653. Springer, 77–97. https://doi.org/10.1007/978-3-642-03013-0_5
- [33] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2010. Four Trends Leading to Java Runtime Bloat. *IEEE Software* 27, 1 (2010), 56–63. <https://doi.org/10.1109/MS.2010.7>
- [34] Nick Mitchell and Gary Sevitsky. 2007. The causes of bloat, the limits of health. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 245–260. <https://doi.org/10.1145/1297027.1297046>
- [35] Vishwath Mohan, Per Larsen, Stefan Brunthaler, Kevin W. Hamlen, and Michael Franz. 2015. Opaque Control-Flow Integrity. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society. <https://www.ndss-symposium.org/ndss2015/opaque-control-flow-integrity>
- [36] musl 2019. musl. <https://www.musl-libc.org/>. Accessed: 2019 June 15.
- [37] Robert Muth, Saumya K. Debray, Scott Watterson, and Koen De Bosschere. 2001. Alto: A Link-time Optimizer for the Compaq Alpha. *Softw. Pract. Exper.* 31, 1 (Jan. 2001), 67–101. [https://doi.org/10.1002/1097-024X\(200101\)31:1<67::AID-SPE357>3.0.CO;2-A](https://doi.org/10.1002/1097-024X(200101)31:1<67::AID-SPE357>3.0.CO;2-A)
- [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 (ISMM '10)*. ACM, New York, NY, USA, 31–40. <https://doi.org/10.1145/1806651.1806657>
- [39] Nicholas Nethercote and Julian Seward. 2007. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, New York, NY, USA, 89–100. <https://doi.org/10.1145/1250734.1250746>
- [40] nginx 2019. nginx. <https://nginx.org/>. Accessed: 2019 June 10.
- [41] Ben Niu and Gang Tan. 2014. Modular control-flow integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 577–587. <https://doi.org/10.1145/2594291.2594295>
- [42] Ben Niu and Gang Tan. 2015. Per-Input Control-Flow Integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 914–926. <https://doi.org/10.1145/2810103.2813644>
- [43] Node.js 2018. Node.js. <https://nodejs.org/en/>. Accessed: 2018 Apr 24.
- [44] norvig 2019. norvig. <https://norvig.com/big.txt>. Accessed: 2019 June 12.
- [45] openssh 2019. openssh. <https://www.openssh.com/>. Accessed: 2019 June 11.
- [46] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi. 2004. Pinpointing Representative Portions of Large Intel #174; Itanium #174; Programs with Dynamic Instrumentation. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*. 81–92. <https://doi.org/10.1109/MICRO.2004.28>
- [47] PAX ASLR 2003. PaX Address Space Layout Randomization. <https://pax.grsecurity.net/docs/aslr.txt>. Accessed: 2018 May 7.
- [48] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.
- [49] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 869–886. <https://www.usenix.org/conference/usenixsecurity18/presentation/quach>
- [50] RAZOR: A Framework for Post-deployment Software Debloating 2019. RAZOR: A Framework for Post-deployment Software Debloating. <https://www.cc.gatech.edu/~hhu86/papers/razor.pdf>.
- [51] ROPgadget 2018. ROPgadget v5.4. <https://github.com/JonathanSalwan/ROPgadget>. Accessed: 2018 May 7.
- [52] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)*. USENIX, Boston, MA, 309–318. <https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany>
- [53] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the 2005 USENIX Annual Technical Conference, April 10-15, 2005, Anaheim, CA, USA*. USENIX, 17–30. <http://www.usenix.org/events/usenix05/tech/general/seward.html>
- [54] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson (Eds.). ACM, 552–561. <https://doi.org/10.1145/1315245.1315313>
- [55] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 329–339. <https://doi.org/10.1145/3238147.3238160>
- [56] sshdCve 2019. sshdCve. <https://nvd.nist.gov/vuln/detail/CVE-2015-7547>. Accessed: 2019 June 12.
- [57] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. 2005. Link-time binary rewriting techniques for program compaction. *ACM Trans. Program. Lang. Syst.* 27, 5 (2005), 882–945. <https://doi.org/10.1145/1086642.1086645>
- [58] Bjorn De Sutter, Ludo Van Put, Dominique Chanet, Bruno De Bus, and Koen De Bosschere. 2007. Link-time compaction and optimization of ARM executables. *ACM Trans. Embedded Comput. Syst.* 6, 1 (2007), 5. <https://doi.org/10.1145/1210268.1210273>
- [59] TensorC 2017. Tensorflow for C. https://www.tensorflow.org/install/install_c. Accessed: 2017 Feb 5.
- [60] Wiem Tounsi and Helmi Rais. 2018. A survey on technical threat intelligence in the age of sophisticated cyber attacks. *Computers & Security* 72 (2018), 212–233. <https://doi.org/10.1016/j.cose.2017.09.001>
- [61] Minh Tran, Mark Etheridge, Tyler Bletsch, Xuxian Jiang, Vincent Freeh, and Peng Ning. 2011. On the Expressiveness of Return-into-libc Attacks. In *Recent Advances in Intrusion Detection*, Robin Sommer, Davide Balzarotti, and Gregor Maier (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 121–141.
- [62] Victor van der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A Tough Call: Mitigating Advanced Code-Reuse Attacks at the Binary Level. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*. IEEE Computer Society, 934–953. <https://doi.org/10.1109/SP.2016.60>
- [63] wikipedia 2019. wikipedia. https://en.wikipedia.org/wiki/Main_Page. Accessed: 2019 June 10.
- [64] wrk 2019. wrk. <https://github.com/wg/wrk>. Accessed: 2019 June 10.

- [65] Guoqing Xu and Atanas Rountev. 2010. Detecting Inefficiently-used Containers to Avoid Bloat. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 160–173. <https://doi.org/10.1145/1806596.1806616>
- [66] Guoqing (Harry) Xu. 2012. Finding reusable data structures. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21–25, 2012*, Gary T. Leavens and Matthew B. Dwyer (Eds.). ACM, 1017–1034. <https://doi.org/10.1145/2384616.2384690>
- [67] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2010. Finding low-utility data structures. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5–10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 174–186. <https://doi.org/10.1145/1806596.1806617>
- [68] Guoqing (Harry) Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7–11, 2010*, Gruia-Catalin Roman and Kevin J. Sullivan (Eds.). ACM, 421–426. <https://doi.org/10.1145/1882362.1882448>
- [69] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. 2016. CREDAL: Towards Locating a Memory Corruption Vulnerability with Your Core Dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24–28, 2016*, Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (Eds.). ACM, 529–540. <https://doi.org/10.1145/2976749.2978340>
- [70] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, and Wei Zou. 2013. Practical Control Flow Integrity and Randomization for Binary Executables. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19–22, 2013*. IEEE Computer Society, 559–573. <https://doi.org/10.1109/SP.2013.44>
- [71] Irene Zhang, Tyler Denniston, Yury Baskakov, and Alex Garthwaite. 2013. Optimizing VM Checkpointing for Restore Performance in VMware ESXi. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. USENIX Association, Berkeley, CA, USA, 1–12. <http://dl.acm.org/citation.cfm?id=2535461.2535463>
- [72] Mingwei Zhang and R. Sekar. 2013. Control Flow Integrity for COTS Binaries. In *Proceedings of the 22Nd USENIX Conference on Security (SEC'13)*. USENIX Association, Berkeley, CA, USA, 337–352. <http://dl.acm.org/citation.cfm?id=2534766.2534796>
- [73] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. 2006. Using Branch Correlation to Identify Infeasible Paths for Anomaly Detection. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-39 2006), 9–13 December 2006, Orlando, Florida, USA*. IEEE Computer Society, 113–122. <https://doi.org/10.1109/MICRO.2006.48>