Saffire: Context-sensitive Function Specialization against Code Reuse Attacks

Shachee Mishra
Stony Brook University
shmishra@cs.stonybrook.edu

Michalis Polychronakis
Stony Brook University
mikepo@cs.stonybrook.edu

***Abstract*—The sophistication and complexity of recent exploitation techniques, which rely on memory disclosure and whole-function reuse to bypass address space layout randomization and control flow integrity, is indicative of the effect that the combination of exploit mitigations has in challenging the construction of reliable exploits. In addition to software diversification and control flow enforcement, recent efforts have focused on the complementary approach of code and API specialization to restrict further the critical operations that an attacker can perform as part of a code reuse exploit.**

In this paper we propose *Saffire*, a compiler-level defense against code reuse attacks. For each calling context of a critical function, Saffire creates a specialized and hardened replica of the function with a restricted interface that can accommodate only that particular invocation. This is achieved by applying *static argument binding*, to eliminate arguments with static values and concretize them within the function body, and *dynamic argument binding*, which applies a narrow-scope form of data flow integrity to restrict the acceptable values of arguments that cannot be statically derived. We have implemented Saffire on top of LLVM, and applied it to a set of 11 applications, including Nginx, Firefox, and Chrome. The results of our experimental evaluation with a set of 17 real-world ROP exploits and three whole-function reuse exploits demonstrate the effectiveness of Saffire in preventing these attacks while incurring a negligible runtime overhead.

1. Introduction

Despite the continuous deployment of exploit mitigations, such as address space layout randomization (ASLR) [54] and control flow integrity (CFI) [1], code reuse [20, 37, 51, 52, 64] is still the most widely used technique for achieving arbitrary code execution through the exploitation of memory corruption vulnerabilities.

The unpredictability introduced by ASLR can be lifted by leveraging a memory disclosure vulnerability to leak the base address of code pages—and even scan them dynamically [66]—to pinpoint the addresses of instruction sequences that can be reused to construct a return-oriented programming (ROP) [64] attack payload. The confinement of indirect control flow transfers to only legitimate targets enforced by CFI restricts significantly the instruction sequences that an attacker can reuse, but as recent works have shown, successful exploits can still be constructed using solely legitimate code fragments and control flow transfers. This can be achieved by transferring control to legitimate call sites in the middle of functions [13, 23, 31, 73], reusing whole functions [19, 62], or through other more advanced techniques, depending on the vulnerability [12, 61].

In addition to software diversification and control flow enforcement, recent efforts in the defense front have also focused on alternative approaches that can further strengthen existing mitigations. In particular, the broader area of attack surface reduction has seen a renewed interest [33, 34, 42, 46, 47, 49, 50, 57, 69, 77, 78, 79], given the benefits of removing unneeded code or functionality. Besides neutralizing any (still undiscovered) vulnerabilities in the unneeded code, removing code from a process' address space means that i) fewer instruction sequences are available to an attacker for building ROP payloads (although the remaining ones are often still enough), and ii) CFI enforcement is simplified due to fewer indirect branches and branch targets that need to be checked.

Shared libraries, in particular, contribute a large amount of unused code, as applications typically import only a fraction of the functions exported by each library. Several *library customization* techniques thus focus on removing unneeded (i.e., non-imported) functions from loaded libraries [2, 50, 57, 69]. However, several critical functions (e.g., related to memory allocation, process management, and file system operations) are unlikely to be removed, as they are typically needed by the application itself.

To mitigate this problem, library customization can be complemented by *API specialization* [49], which restricts the interface of the remaining critical API functions according to the actual needs of a given program. The main intuition behind API specialization is that essential functionality for malicious code (e.g., giving "execute" permission to some memory area that contains second-stage shellcode) may not be required by the application.

Shredder [49] implements this approach by statically analyzing Windows applications at the binary level, and deriving application-wide policies for critical system functions. Although Shredder's policies are effective in blocking ROP exploits in many applications, there is significant room for improvement due to two major limitations: i) statically deriving the possible values of function arguments at the binary level is complicated due to the imprecision of code disassembly and control flow graph extraction, and ii) the values of many arguments cannot be statically derived at all, as they become known only at run time.

Aiming to broaden the scope of API specialization and further restrict the extent to which an attacker can influence the arguments of critical system functions, in this paper we present *Saffire*, a compiler-level approach for context-sensitive function specialization and hardening. For each call site of a critical function, Saffire creates a

© 2020, Shachee Mishra. Under license to IEEE.
DOI 10.1109/EuroSP48549.2020.00010

custom version of the function with a restricted interface that can accommodate the needs of only that particular call site. This is achieved by performing inter-procedural backwards slicing and data flow analysis in a best-effort way to identify the source of each argument, and concretize its value (or set of possible values) in the body of the customized function.

Besides static arguments, however, a key novelty of Saffire is that it also protects *dynamic* arguments, such as file descriptors, memory addresses, and user-supplied inputs, which become known only at runtime—based on our evaluation, about half of the arguments of critical functions fall into this category, and are ignored by previous works. To protect such arguments, we introduce *dynamic argument binding*, which applies a narrow-scope form of data flow integrity to restrict the argument values permitted at runtime for a given call site, to only those that were originally assigned at their respective initialization points.

We implemented a prototype of Saffire for Linux applications as a transformation pass on top of LLVM, and experimentally evaluated it with a set of real-world code reuse exploits and applications, which demonstrate its effectiveness and practicality. In particular, the lightweight nature of its policy checks introduces negligible performance overhead, while it blocks all 17 ROP payloads tested in all but two of the 11 applications considered. We also demonstrate how Saffire can protect against sophisticated whole-function reuse attacks that aim to bypass CFI, by evaluating it against three previously published proof-of-concept exploits against Nginx [23], Firefox [62], and Chrome [19].

In summary, we make the following main contributions:

- We propose a compiler-level defense-in-depth mitigation against code reuse attacks, which introduces *static* and *dynamic* argument binding to restrict the values that can be passed to critical API functions.
- We implemented *Saffire*, a prototype of the proposed approach on top of LLVM, and applied it to a set of 11 applications, including Nginx, Firefox, and Chrome.
- We experimentally evaluated Saffire with a set of 17 real-world ROP exploits and three proof-of-concept whole-function reuse exploits, and demonstrate its effectiveness in preventing these exploits while incurring negligible runtime overhead.

2. Background and Motivation

The main motivation behind the proposed approach is the hypothesis that the system calls made by an application are semantically different from the ones made by malicious code. Specializing the exposed system API according to an application's actual needs can thus hinder attacks that rely on functionality that is not used by the application. In Linux, the system call interface is exposed to user programs through the C Standard Library (libc). Although libc exports more than 1,400 functions, programs typically use only a fraction of them [49, 50, 57]. More importantly, even for the functions that are used, only part of their code is executed, depending on the subset of their functionality actually used by the application [49].

2.1. Code Reuse Attacks and Function Reuse

Since their introduction in the form of return-to-libc [20], code reuse attacks have been continuously evolving in response to an increasing number of deployed exploit mitigations. Return-oriented programming (ROP) [37, 64] and its variations [11, 15, 66], are currently the de facto method of achieving arbitrary code execution through the exploitation of memory corruption vulnerabilities. Typical ROP exploits consist of a first-stage payload that aims to bypass non-executable memory protections and enable the execution of a second-stage shellcode. To achieve its goal, the first phase relies on existing instructions (ROP gadgets) to interact with the OS (through system calls), e.g., for allocating executable memory or accessing the file system.

Recently, there has been an emergence of more advanced code reuse attacks that aim to bypass control flow integrity (CFI) [1] defenses by reusing longer instruction sequences or even whole functions that do not violate the enforced control flow policy. Recent developments [13, 23, 62] have shown that instead of using just a set of instruction sequences, using full functions as gadgets can lead to arbitrary code execution.

2.2. Challenges of API-level Specialization

Some code debloating techniques focus on removing unused functions from imported libraries, reducing this way the amount of code that can potentially take part in a code reuse attack. Such library debloating techniques can be applied at either the binary [2, 50] or the source code [57] level. Upon the application of library debloating, API specialization [49] moves one step further and restricts the allowable argument values that can be passed to the remaining system API functions that cannot be removed by library debloating. Instead of removing code, a runtime interposition layer verifies the passed arguments against a statically determined set of allowable values. This set is created by analyzing the binary to identify library functions and control flows within them that are required by the application, and then in a best-effort way derive the argument values used across their call sites.

Binary vs. Source Code Analysis: Prior API specialization approaches [49] face several challenges. First, operating at the binary level impedes their ability of accurately finding all valid control flow paths, and more importantly, performing accurate data flow analysis to identify statically determined argument values. Due to the small number of general purpose registers, their values are constantly overwritten, hindering argument extraction even further. In this work, we perform more accurate and comprehensive data flow analysis at the source code level, to expand the set of known argument values.

Context-sensitive Policies: Second, finding known arguments across all calling contexts of a function is not always possible. Prior solutions do not distinguish one call site of a library function from another, and the policies generated are universally applied for the whole program. Although this makes runtime policy enforcement simpler, it misses many enforcement opportunities. For example, in case a certain argument has its value known across all but one call site, an application-wide policy would still consider it unknown.

TABLE 1: Number of known argument values across all call sites of some libc functions for Nginx.

Library Function	Call	Known Values			
	Sites	Arg1	Arg2	Arg3	Arg4
`open64()`	21	0	19	18	21
`write()`	23	16	11	12	–
`read()`	3	0	0	1	–
`pread64()`	6	0	0	3	4

A few examples of such cases are shown in Table 1, which reports the number of known argument values across all call sites for some of the libc functions used by Nginx. For instance, `open64()` is invoked by 21 call sites, and although its fourth argument is known across all call sites, the same is not true for the second and third arguments, the values of which are known for all but two and three contexts, respectively—application-wide policies thus cannot be generated for them. This opens up the possibility for attackers to reuse any of the available call sites with their desired argument values, even though for most call sites this could be prohibited. Creating application-wide policies thus misses the opportunity to enforce stricter policies for certain arguments that are known only in some (but not all) calling contexts. In this work, we address this issue by deriving and enforcing context-sensitive policies, which restrict further the possibilities of call site reuse as part of malicious code.

Dynamic Function Arguments Finally, and more importantly, a significant challenge faced by API-level specialization is that the actual values for many function arguments cannot be derived statically at all. User inputs, environment variables, file names, memory addresses, and other types of arguments will only become available at run time. Current systems mark arguments that hold such dynamic values as unknown. Consequently, exploit code can still rely on manipulating those arguments despite the policies enforced by systems like Shredder [49]. In this work, we address this issue by introducing a form of narrow-scope data flow integrity for dynamic function arguments to prohibit attackers from supplying arbitrary values as part of code reuse exploits. As demonstrated by our experimental evaluation results, our approach roughly doubles the number of arguments that can be protected, compared to using only statically-derived policies.

2.3. Threat Model

We assume an attacker who is able to hijack the control flow of a process and execute malicious ROP code (and possibly second-stage shellcode), which interacts with the OS to achieve the attacker's end goal (e.g., remote control, DLL injection, malware installation). In that sense, data-only attacks (e.g., modifying a user authentication variable without using any system calls), are out of scope.

We also assume that common exploit mitigations, such as non-executable memory and ASLR, have been deployed on the system. Although Saffire offers the same protection irrespective of these defenses, as a defense-in-depth approach, it is mostly meant to be used in combination with other defenses to collectively raise the bar for successful exploitation, and prevent circumvention. Similarly, Saffire's protection becomes meaningful only after existing software debloating techniques such as library debloating [2, 50, 57] have first been applied. As Saffire's policy enforcement is performed by user-level code, we assume that fine grained CFI is employed, so that an attacker cannot simply bypass policy checks (e.g., by jumping over the check, or even invoking system calls directly). Alternatively, other hook protection mechanisms such as API checkpointing [53] can be employed.

3. Function Specialization

Our main goal is to create multiple specialized versions of API functions that can only be called from specific locations within the application. Exploit code can use any of the functions in the address space to carry out malicious operations not intended by the application. Specializing functions according to their invocation points reduces the flexibility of an attacker in reusing them.

Control flow integrity confines the possible invocation points of API functions to only legitimate call sites within the program. On top of CFI, Saffire then confines further the possible argument values that can be supplied to a function to the absolutely needed ones by a given *calling context*. For direct function invocation, the calling context corresponds to the call site of a given function. For indirect function invocation through function pointers, the calling context corresponds to the union of all possible call sites of a given function pointer (more details on this case are provided in Section 4.1). For a given calling context, Saffire creates a unique custom function that will be used solely by that particular context.

Saffire's custom functions "bind" arguments to their corresponding calling contexts. Every call to a library function is now routed through a custom function wrapper which first verifies the supplied arguments, and then invokes the original library function after the verification is successful. We should note that the use of wrapper functions is not mandatory, and an alternative design could just create as many specialized copies of the original function as needed, and then completely remove the original function from the address space (e.g., using existing library debloating techniques [57]). We just chose this approach to simplify our engineering effort. From a security perspective, both approaches are equivalent, since the enforced CFI policy still confines the allowable entry points to only the original call sites of the program, i.e., an attacker cannot bypass the wrapper and jump straight to the library function.

Creating specialized versions of library functions requires precisely identifying as much of the state of their arguments as possible for each calling context. We observe that argument values broadly fall into two categories: i) static, deterministic values that can be derived from analyzing the code, and ii) values that are dynamically generated at runtime. In the example below, `ptr` and `size` can only be known at run time, while `prot` can be determined statically, as it is a hard-coded value:

```
char *ptr = mmap(..);         // Dynamic argument
int size  = getpagesize();    // Dynamic argument
int prot  = 2;                // Static argument
mprotect(ptr, size, prot);
```

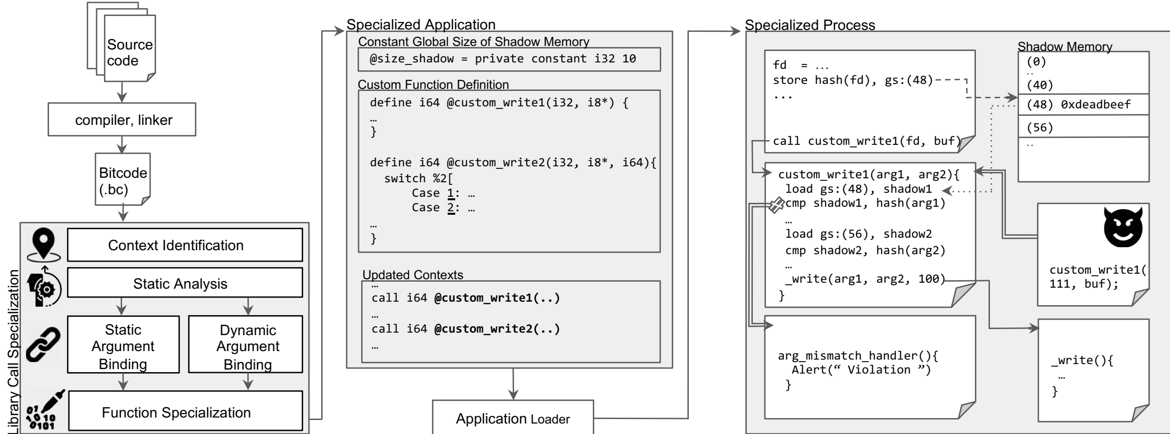

Figure 1: Overview of the proposed approach. Saffire identifies the call sites of specific library functions and performs static analysis to instrument the code with static and dynamic argument binding in the combined bitcode file. At runtime, dynamic values are hashed and copied into a shadow memory at creation time, and are compared with the supplied arguments at verification time inside the customized function.

Static and dynamic arguments have to be treated in different ways while specializing functions. In both cases, static code analysis can be used to identify the point where the values are generated. For static arguments, their actual value (or set of possible values) can be enforced by constructing a set of corresponding static rules, to which we refer as *static argument binding*. For dynamic arguments, however, this is not possible, since their values may depend on external factors, such as user input and OS state. Instead of leaving such arguments unprotected (as done by previous API specialization approaches [49]), we have developed an alternative protection mechanism called *dynamic argument binding*. Using a narrow-scope form of data flow integrity, dynamic argument binding restricts the argument values permitted at runtime to only those that were created at their respective initialization points.

Figure 1 shows an overview of our approach. The core of Saffire's static code analysis and instrumentation is performed as an optimization pass in LLVM, which operates on a combined bitcode file generated after link time optimization. In this example, the specialized application contains two call sites of the `write()` function, for which Saffire creates two customized variants. According to the number of *statically known* arguments, the actual number of input arguments to these functions may be different. To perform dynamic argument binding, the specialized process keeps a 64-bit hash of dynamic argument values (at the time of their creation) to a shadow memory area, which are then used to verify the actual input arguments. In this example, `custom_write1()` first compares the hash values of its two arguments with their respective values in the shadow memory, and proceeds to call `write()` only if the verification is successful. When attackers attempt to reuse the function with a different set of arguments, the verification fails and control is redirected to a handler.

3.1. Static Argument Binding

For a given calling context, an argument is considered *known* if all its possible values can be determined statically. There are three main types of known arguments:

1) Hard-coded arguments: Arguments such as flags or buffer sizes, are often known in advance.
2) Value sets: Multiple control flow paths may set different values to a given argument variable, all of which can still be identified statically.
3) Value ranges: For certain calling contexts involving loops, the start, limit, and increment values of an integer variable can be determined statically, resulting in a range of possible values.

Static argument values are defined somewhere on the path leading up to `call` instructions. The analysis phase of Saffire follows these control paths backwards, starting from the call site, and tries to identify the value(s) assigned to a given argument. The analysis continues over function boundaries using inter-procedural analysis when a tracked variable is passed from a previous caller to the current function. We leverage the control flow information available at the intermediate representation level to find predecessors for function calls. We also use address-taken information to find specific instances of functions assigned to pointers. For each usage of a given function, the analysis continues traversing backwards across functions until either the value is found or we reach the beginning of `main()`. For example, when analyzing the call site of `execve()` in Nginx, the analysis reaches `main()`, concluding that the first `execve()` argument is a user-provided value.

The compiler front-end creates a set of object files, which are then linked to create the final executable after several rounds of optimizations. A particularly important one for our purposes is *constant propagation*, which identifies constants assigned to variables, propagates them through the control flow graph, and substitutes their values at the points where variables are used. This optimization reduces the stack space requirements of the process, while also reducing unnecessary arithmetic computations at runtime.

Saffire's analysis goes deeper to find *sets* of possible values for each calling context. By performing our analysis on top of constant propagation, we are able to expand the

TABLE 2: Static and dynamic argument binding examples.

<table>
<tr><th></th><th>Original</th><th>Specialized</th></tr>
<tr><td>Static Binding: Single Arg.</td><td>a = 2;
func1(a,NULL);</td><td>custom_func1();

custom_func1(){
   func1(2, NULL);
}</td></tr>
<tr><td>Static Binding: Set of Args.</td><td>if(x)
  a = 2
else
  a = 3
func2(a);</td><td>if(x)
  a = 2
else
  a = 3
custom_func2(a);

custom_func2(int a){
  if(a==2 || a==3)
      func2(a);
}</td></tr>
<tr><td>Dynamic Binding</td><td>fd = open(....);
func3(fd);</td><td>fd=open();
shadow[1] = hash(fd);
custom_func3(fd);

custom_func3(int fd){
 if(hash(fd) == shadow[1])
    func3(fd);
}</td></tr>
</table>

set of possible known values further. As an example, the first two parts of Table 2 list two simple call sites and their specialized versions. In the upper part, constant propagation would hard-code the integer value 2 passed as an argument to `func1`, but cannot do so for `func2` (middle part), as argument `a` may take one of two possible values. Saffire derives the set of possible values for this calling context and enforces a corresponding policy beyond what constant propagation alone could achieve.

Once all static argument binding policies have been generated, Saffire creates instances of corresponding specialized functions and adds their definitions in the program. To enforce static argument binding, these new functions either verify at runtime the passed arguments according to a set of known values (for value set or value range arguments, e.g., middle row in Table 2), or omit the arguments altogether (for hard-coded arguments, e.g., upper part in Table 2). Finally, all call sites are adjusted to invoke the newly added specialized versions.

3.2. Dynamic Argument Binding

Although many arguments can be statically derived, based on our experimental evaluation, we observed that about half of the arguments across the tested library functions become known only at run time. These mostly correspond to user input, memory or file operands, environment variables, and other OS-related states. As an example, the value of `fd` in `func3()` in the bottom part of Table 2 will depend on the file descriptor number the OS will pick. To ensure that this particular call site will not accept arbitrary values for argument `fd`, dynamic argument binding ensures at runtime that the only acceptable value will be the one returned by `open()`.

Dynamic argument binding is in essence a narrow-scope form of data flow integrity, enforced between variable definitions and their use as input arguments. These dynamic variables typically remain live for short periods, and are relatively easier to track compared to full-program data flow integrity. Similarly to the analysis used for static argument binding, we perform backwards slicing to determine the instruction where the argument value was defined or last

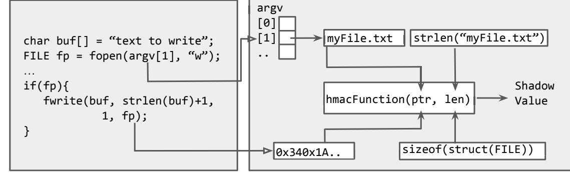

Figure 2: Each shadow argument corresponds to the HMAC of the argument's actual value. Here, for `fwrite()`, the input FILE struct is read from memory and is passed to the HMAC function along with the size of the struct.

modified (whichever is later), and instrument that location to generate a hash of the data and save it in a *shadow memory* location. The specialized function reads the 64-bit hash from the shadow memory and verifies the input argument against it by hashing the received input argument, before proceeding with the actual library call.

Each such dynamically defined variable has a *pre-defined* index reserved for it in the shadow memory. As this transformation is performed at compilation time, these indexes are determined and inserted into the binary statically. To ensure that the attacker is unable to write arbitrary values to shadow variable, the shadow memory area is kept isolated (see Section 3.2.2). This method can be used for memory addresses, file descriptors, pointers to variables, and other arguments of a similar nature.

Regarding memory address arguments, although these are mostly assigned at run time, sometimes they receive a NULL or other constant value that can be determined statically (e.g., as shown in the upper part of Table 2). This means that for some call sites of a given function, we are able to extract these otherwise dynamic arguments during static analysis, in which case they are protected through static argument binding, while for the rest they are protected through dynamic argument binding. This "hybrid" treatment of arguments cannot be applied for pointers and addresses that may be reassigned or have their values change from declaration to usage time if they are passed to other functions in between. A more sophisticated analysis could track the use of pointers across multiple functions and derive better bindings, but we leave this optimization as part of our future work.

An overview of the shadow argument generation process is shown in Figure 2. We use an HMAC (hash-based message authentication code) with SHA-256 to generate a 64-bit shadow value for dynamic arguments. The secret key for the HMAC is generated at start-up time and is saved at index 0 in the shadow memory (a dedicated register could also be used for this purpose). In this example, the input to `fopen()` is a user-supplied filename. The pointer to the filename and its length are supplied to the HMAC function for generating the argument's shadow value. This value is compared with the shadow copy that was generated using the same process at the prior point of the argument's definition (not shown here).

3.2.1. Function Pointers. When function pointers are used, the same originating variable can be used as an argument to any of the potential target functions. In such cases, every such definition must be bound to every target function. This is explained in Figure 3, where the

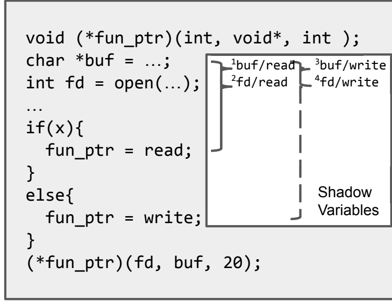

Figure 3: For function pointer arguments, dynamic binding creates shadow copies for every pair of originating variable and target function. Here, read and write are both possible targets of func_ptr, and the buf and fd variables are both passed as arguments, resulting in four different shadow variables, one for each pair.

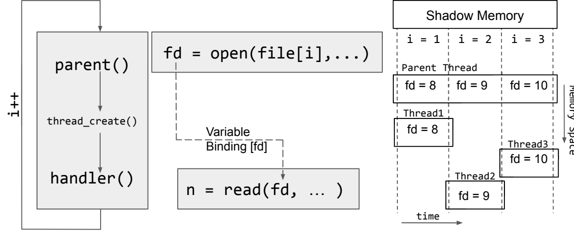

Figure 4: Shadow memory for multi-threaded applications. In the parent process, fd is returned by open(), which is then passed to read() in handler(). There is only one path, thus one shadow variable for fd. Newly created threads contain an up-to-date shadow copy of fd, as their shadow memory is initially copied from the parent.

definitions of fd and buf are assigned two shadow locations each, one for each target (read() and write()). Although the same shadow copy could be used for all target functions, to simplify the implementation, we choose to assign a different shadow copy for each target function.

3.2.2. Shadow Memory Protection. To prevent the attacker from tampering with the shadow copies, the shadow memory must be isolated. Intel's Memory Protection Keys (MPK) technology [18] provides an interface for lightweight user-level memory protection switching for individual pages, which can be used for in-process memory isolation [71]. Despite the fact that MPK is only available in very recent processors, managing permissions per page means that we should either devote one page per variable, or group multiple copies into the same page.

To avoid the ensued complexity and to maximize compatibility, we opted for the use of memory hiding to isolate the shadow memory. Similarly to previous works [43], we devote one of the unused segment registers for keeping the base address of the shadow memory. All related operations use the segment register as part of the indexing calculation without ever leaking its value into memory, keeping this way the location of the shadow memory secret from the attacker.

We should stress that the choice of the isolation mechanism used to protect the shadow memory is completely orthogonal to Saffire's operation, and if reliance on memory hiding is a concern, then MPK-based isolation [71] or any other similar mechanism can be used instead.

3.2.3. Multi-threaded Programs. For multi-threaded programs, every thread has its own shadow memory, which is updated according to variable definitions. When a new thread is created, it allocates memory to be used as its shadow memory. To ensure that each thread has access to variables declared by its parent process, the shadow memory of the parent is copied to thread's shadow memory during thread creation, as shown in Figure 4. On every iteration, the parent process always uses the same shadow memory, while every thread gets a new shadow memory instance, initially copied from the parent.

4. Implementation

We have developed Saffire, a prototype of the proposed approach for Linux 64-bit ELF binaries. Saffire is implemented as a transformation pass on top of LLVM-8, and has been tested on 64-bit Ubuntu 19.04. The input to Saffire is a combined bitcode file created after link-time optimization.

4.1. Static Argument Binding

Saffire performs backwards data flow analysis at the IR level, starting from every call site of each protected API function. The process begins by identifying the invocation points of functions, which include i) direct calls, and ii) indirect calls through function pointers.

Note that in the latter case, there is no need to determine all possible targets of indirect calls. Instead, for a given function assigned to a function pointer, all its possible uses (call sites) are identified through LLVM's def-use chains. The custom function assigned to the function pointer is then specialized according to the *union* of all possible call sites, which we treat as its calling context. In the example below, this means that custom_foo() and custom_bar() will accept both 100 and 200 as allowable argument values.

```
int ( *fp )(int);
if (x) {
    fp = foo;   //assigned to custom_foo(int)
}
else{
    fp = bar;   //assigned to custom_bar(int)
}

fp(100);      //calls custom_foo or custom_bar
...
fp(200);      //calls custom_foo or custom_bar
```

The type of each argument is identified using LLVM's Value::getType(). Integer arguments are extracted directly to obtain their current value. If an argument's value is a constant, dyn_cast<ConstantInt>() returns true. At this time, we also start preparing the specialized function for this calling context. Constant values are never passed via arguments, but are directly used within the body of the specialized function. NULL values are detected using dyn_cast<ConstantPointerNull>() and are also removed from the input arguments.

4.2. Dynamic Argument Binding

For each dynamic argument, we first identify the instruction where it was defined or last modified (using the same inter-procedural analysis used in static argument binding), and instrument the code at this point to create its shadow value. At invocation time, this value will be compared with the HMAC of the current input inside the specialized function, before proceeding to the library call.

Struct members in LLVM-IR are accessed using the `getelementptr` instruction. All def-use chains of the structure are followed to find the last instruction where the struct is written to. A getelementptr followed by a store indicates that a struct member is being written to. Dynamic binding follows right after the instruction.

When the point of last assignment to a pointer is followed by another function call with this variable as an argument, we mark the argument as unknown. This is because its target address may be updated by the callee, and in our current implementation we do not perform forward tracking across functions. In the example below, `buf` is initialized to `buf_old` as its last modified value before `write()`, but `func(buf)` is called between the definition and use of `buf`. This function may change the value of `buf`, and thus we mark `buf` as unknown.

```
char *buf = buf_old;
int n = func(buf);
write(fd, buf, 20);
```

4.2.1. Shadow Memory. The integrity of dynamic argument binding relies on ensuring that only pre-specified writes are allowed to the shadow memory. An attacker must not have the ability to overwrite this memory with malicious variable values and bypass the verification. In our current implementation we rely on memory hiding to prevent the attacker from accessing the shadow memory. We use segment register `%gs`, which is not used in user space by any binaries or libraries, to store the base address of the shadow memory region. The same can be done with any other register not used by the applications in consideration, but we choose `%gs` to maximize compatibility with existing libraries and programs.

The size of the shadow memory can be calculated in advance based on the number of arguments that require dynamic protection, and the location of the region is selected at random within the vast process address space. We assign one 64-bit slot for each argument that needs protection. Then the segment descriptor is allocated and its various fields (e.g., base address and limit) are initialized accordingly. Finally, using the system call `modify_ldt()`, this segment descriptor is inserted into the Local Descriptor Table. As `modify_ldt()` does not have a libc wrapper, this call is directly made using `syscall(__NR_modify_ldt)`.

Once the segment selector is loaded to `%gs`, it remains there unmodified for the entire duration of the process. Every dynamic variable for every calling context has a pre-defined index in the shadow memory. The target index is specified using `%gs:(offset)`. A linear address calculation module extracts the base address from the descriptor indexed by `%gs` and adds the appropriate offset to obtain the address of a given argument's shadow copy.

4.2.2. Shadow Arguments. We use Hash-based Message Authentication (HMAC) using SHA-256 to create a 64-bit digest of the entire data object passed as an argument. For arguments passed by value (e.g., integer arguments), the hash is directly created. For arguments passed by reference (e.g., strings, arrays, lists, structs), length information depends on the particular object type (strings are NULL-terminated, the size of structs is known based on their type, and so on). For complex objects containing pointers to other second-level objects, only the parent object (including all pointer values) is hashed. As we discuss in Section 6, this is not a major limitation, as the security-critical API functions we are concerned with (see Section 5.1) involve simple objects, such as strings, pointers to raw memory, and file descriptors. Recursive hashing can be implemented to provide protection for complex objects if needed.

In our current prototype, the HMAC implementation is part of an external library that is dynamically linked. Functions to create, store, and verify the HMAC of an input argument thus involve making a library call. Even with these additional library calls, our experimental evaluation shows that the overhead remains negligible.

4.3. Function Specialization

The static analysis phase returns a map, in which each key corresponds to an argument and each value to a set of possible argument values or an identifier to the instruction where dynamic argument binding is performed. Specialized functions are created by i) removing single assignment variables from input arguments, ii) adding verification checks for variables with multiple possible values, and iii) performing dynamic binding. The specialized function is inserted into the IR and a `call` instruction to it is added immediately after the original call to the library function. Next, all the uses of the return value of the original call are replaced with the return value of the specialized function using `replaceAllUsesWith()`. Finally, the original call instruction is removed using `eraseFromParent()`.

To prevent attackers from jumping over the specialized wrapper functions, we enable LLVM's CFI support. This ensures that attackers cannot divert execution to arbitrary addresses within the text section, and are restricted to only use full functions (e.g., using COOP attacks [62]). We use the `-fsanitize=cfi` option to enable strict cast checks and type checking for indirect, virtual, and non-virtual calls.

Table 3 presents examples of a transformed call to `mmap()` at the source code and IR levels for the three cases mentioned in the previous section. In the first case (Table 3(a)), arguments 1, 3, 4, and 6 have known immediate values, while arguments 2 and 5 remain unknown (i.e., backwards analysis fails to locate their initialization points), and thus `custom_mmap` takes only these two arguments. In the second example, backwards analysis identifies that the possible values for the second argument are $\{2, 3\}$, and thus the wrapper function checks the second argument of `mmap()` against these two values. Finally, the third example presents a simplified version of the IR transformation required for dynamic argument binding of the fifth argument (`offset`). The argument value %11 is hashed and the generated hash is copied into the shadow memory. Inside the custom function, the shadow variable

TABLE 3: Example of function specialization for a call to `mmap()` (a) with four known arguments; (b) with a set of known values {2, 3} for the second argument; and (c) with dynamic argument binding for the fifth argument. The upper part shows the C code after transformation, and the lower part shows the corresponding LLVM IR code.

(a) Static argument binding

```
Calling Context
ptr = custom_mmap(len, offset)

New Function
void* custom_mmap(int length,
        int offset) {
  return mmap(null, length,
    1, 32770, offset, 0);
}
```

```
Calling Context
%17 = call i8* @custom_mmap(
                i64 %16, i32 %11)

New Function
define i8* @custom_mmap(i64, i32){
entry:
  %res = call i8* @mmap(
      i8* null, i64 %0, i32 1,
      i32 32770, i32 %1, i64 0)
  ret i8* %res
}
```

(b) Static binding with set of values

```
Calling Context
ptr = custom_mmap(len, offset)

New Function
void* custom_mmap(int length,
        int offset) {
  switch (length) {
    case 2:
    case 3:
      return mmap(null, length,
        1, 32770, offset, 0);
  }
  return -1;
}
```

```
Calling Context
%17 = call i8* @custom_mmap(
                i64 %16, i32 %11)
New Function
define i8* @custom_mmap(i64, i32){
entry:
 switch i32 %0, label %4 [
    i32 2, label %3
    i32 3, label %3
  ]

; <label>:3:    ;preds =%entry
  %res = call i8* @mmap(
      i8* null, i64 %0, i32 1,
      i32 32770, i32 %1, i64 0)
  br label %4

; <label>:4:   ;preds =%entry,%2
  %6 = phi i32 [ %res, %3 ],
               [ -1, %entry ]
  ret i8* %6
}
```

(c) Dynamic argument binding

```
Calling Context
offset = get_offset()
shadow1 = hash(offset)
ptr = custom_mmap(len, offset)

New Function
void* custom_mmap(int length,
        int offset) {
  if (hash(offset) == shadow1) {
    return mmap(null, length,
      1, 32770, offset, 0);
  }
  return -1;
}
```

```
Calling Context
%11 = call get_offset()
%12 = call hash(i32 %11)
store i64  %12 i32* %gs:(1)

%17 = call i8* @custom_mmap(
                i64 %16, i32 %11)

New Function
define i8* @custom_mmap(i64, i32) {
entry:
  %3 = load i64, %gs:(1)
  %4 = call hash (i32 %1)
  %5 = icmp eq i32 %3, %4
  br i1 %5, label %6, label %7

; <label>:6:    ;preds =%entry
  %res = call i8* @mmap(
      i8* null, i64 %0, i32 1,
      i32 32770, i32 %1, i64 0)
  br label %7

; <label>:7:   ;preds =%entry,%6
  %8 = phi i32 [ %res, %6 ],
               [ -1, %entry ]
  ret i8* %8
}
```

is read to a local variable and the hash of the incoming argument value is compared against it. The library call proceeds only if these two values match. If the comparison fails, the control moves to an argument mismatch handler and returns -1.

5. Evaluation

In this section, we evaluate Saffire in terms of effectiveness against real-world exploits and runtime overhead. All experiments were performed on a system equipped with an Intel Core i7-6700 CPU, 16GB RAM, 256GB SSD, running 64-bit Ubuntu 19.04.

5.1. Data Set

To evaluate the effectiveness of Saffire against realistic attacks, we collected 17 Linux-based ROP exploit samples from Exploit-DB and individual real-world and proof-of-concept exploits—a detailed list of all exploits is provided in the Appendix. Given that implementing the entire malicious code functionality using ROP is quite complex (but not impossible), most of the available ROP exploits use only a few system calls (e.g., to spawn a shell or to enable the execution of a second-stage shellcode). In either case, the ROP code has to interact with the OS by invoking one or more system functions through library calls. As an additional case study, we evaluated Saffire against proof-of-concept COOP [19, 62] and Control Jujutsu [23] exploits, which rely on whole-function reuse to bypass CFI defenses.

We tested Saffire with a diverse set of 11 popular applications, including servers (Nginx, Lighttpd, Vsftpd), various utilities (Gzip, OggEnc, OggDec, PuTTY, Ctags), as well as OpenSSH, Google Chrome, and Mozilla Firefox. The set of ROP exploits used for our evaluation are usually meant for specific applications, but because they all rely on libc functions to interact with OS, we make the worst-case assumption that each of them can be used against each of the applications tested—although an exploit meant for a different application will not work, its ROP payload, and in particular the library calls it makes, can equally well be used against any application.

For most of our analysis (except the overhead analysis), we identified a set of 15 critical system library functions used by in-the-wild and proof-of-concept exploits. In addition to the ROP payloads in our set of exploits (mentioned above), we studied shellcode samples from sources such as Metasploit, Exploit-DB, and academic research, to finalize this set of functions. These calls include `mmap`, `mmap64`, `mprotect`, `open`, `open64`, `access`, `execve`, `fstat`, `read`, `write`, `clone`, `pread64`, `fopen`, `fwrite`, `fseek`. For performance and memory overhead analysis, we perform two rounds of experiments by hardening i) the above 15 functions, and ii) *all libc functions*. Although from a security perspective protecting all functions is not necessary, as most of them cannot carry out harmful operations, we wanted to stress-test our implementation with a worst-case scenario.

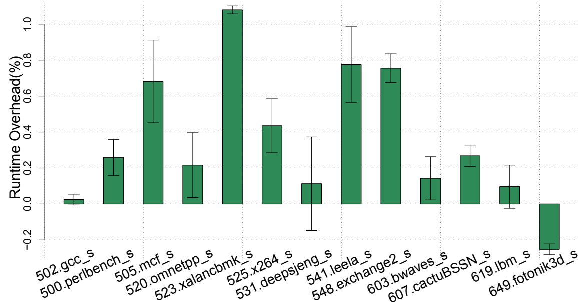

Figure 5: Saffire's runtime overhead for the SPEC2017 benchmarks with *all* libc functions instrumented. The average overhead across all benchmarks was 0.34%.

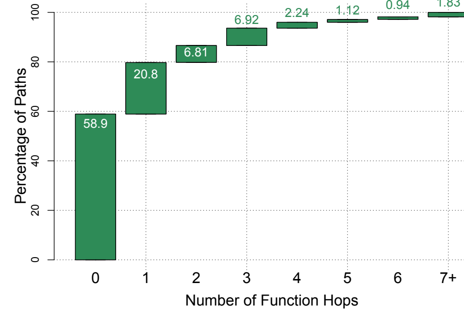

Figure 6: Cumulative distribution of the number of caller functions traversed during backward analysis for reaching the definition of protected arguments. More than half of the arguments are found within the same function.

5.2. Runtime Overhead

Saffire inserts custom functions through interposition, which can impact performance due to the additional indirection and operations performed by the wrapper functions for policy enforcement. To measure this overhead, we conducted two experiments using a set of microbenchmarks and the SPEC2017 benchmarks. We started with a worst-case scenario by stress-testing Saffire with sample programs that repeatedly execute the instrumented library functions with random inputs. Each experiment performed 10 million function calls and results are averaged over five runs for each library function. We perform these tests with no known arguments to ensure worst-case performance. Across all functions, the average performance difference measured with and without instrumentation is less than 3.2%. For instance, for `fwrite()`, the average slowdown was 5.8%, while for `read()` it was 0.35%.

To assess the performance impact of Saffire further, we used the SPEC2017 benchmarks. The SPEC benchmarks do not involve many invocations of the above 15 critical functions, and we thus assumed a worst-case scenario in which *all* libc functions are protected—across all benchmarks, Saffire created custom versions for 304 libc functions (18 functions that do not take any arguments are excluded). Figure 5 shows Saffire's overhead (averaged over multiple runs). The highest overhead of 1.08% was exhibited by `xalancbmk`, while the average overhead across all benchmarks is 0.34%. The infrequent invocation of libc functions compared to the actual computation performed makes Saffire's performance impact quite negligible.

5.3. Inter-procedural Analysis

The static analysis phase of Saffire starts from a calling context and traverses the code backwards to find the assigned value to (or the source of) each argument. In many cases, values are passed as input arguments from previous functions. In such cases, the analysis proceeds backwards to all call sites of the currently analyzed function, and keeps traversing caller functions until either the assigned value is found, or the analysis concludes that it was returned from an unknown function call or passed as a user input.

Figure 6 shows the CDF of the number of caller functions that were traversed to reach the definition of protected arguments. We see that more than half (58.9%) of the argument definitions are found within the body of the same function in which the analyzed call site is located. For about 21% of the arguments, their initialization point is located at the previous caller function, while there are cases where several functions have to be traversed (16 in the worst case).

We should note that the number of paths explored (Y axis) is not the same as the number of arguments. To be precise, although there are 1,837 arguments (for both static and dynamic argument binding) across all calling contexts in the 11 applications, the number of explored paths is 2,182. This is because, depending on the control flow of an application, there can be multiple paths traversed for a given argument and call site.

5.4. Code and Memory Overhead

Function specialization per call site unavoidably introduces additional code for each of the wrapper functions that correspond to the individual call sites. As shown in Table 4, for the larger applications, the number of functions increases by less than 7%. For example, there is an increase of 3% in the number of functions for Nginx. The increase is more pronounced for smaller utility programs, like Gzip. By inspecting the code of Gzip, we found that there are 138 calls to `write()`, for each of which Saffire creates a specialized instance.

Many of these functions, however, happen to be identical (i.e., they perform the same operations and enforce the same argument policies), and thus could be combined into one. This will greatly reduce the number of functions added, while providing the same security benefit. We plan to implement this optimization as part of our future work. As the next column shows, the additional memory needed for storing the hashes of the shadow arguments is negligible.

The overall impact of Saffire's instrumentation is reflected in number of instructions added to the binary. We see that for most applications, the increase is less than 1% in the number of IR statements added. Chrome and Nginx, given their large sizes, have the lowest increase of 0.1% and 0.2%, respectively. The worst increase is observed in OggDec and OggEnc (15% and 45%), which is mostly because these are smaller programs. We also report the time taken by Saffire's pass, which corresponds to the increase in build time for the whole application. We see

TABLE 4: Code and memory increase, time for Saffire's LLVM pass, and number of ROP payloads blocked (out of 17).

Application	# of Functions Added	Size of Shadow Memory per Thread (8-byte units)	# of IR Statements Added	Time Taken to Run Pass (sec)	# of ROP Payloads Blocked
Nginx	58 (3%)	76	1291 (0.2%)	18.2	17
Ctags	27 (2%)	21	926 (0.3%)	8.4	17
Gzip	147 (120%)	39	2142 (8.1%)	3.7	17
Lighttpd	82 (13%)	77	2611 (2.9%)	10.2	12
Vsftpd	15 (6%)	15	784 (2.2%)	6.8	17
PuTTY	51 (4%)	64	2102 (0.8%)	14.2	17
OggEnc	28 (26%)	50	1327 (15.6%)	9.1	17
OggDec	9 (23%)	14	628 (45.3%)	11.3	17
Chrome	92 (1%)	132	3132 (0.1%)	52.5	17
Openssh	63 (11%)	111	2125 (1.3%)	21.2	12
Firefox	42 (10%)	26	684 (0.4%)	28.1	17

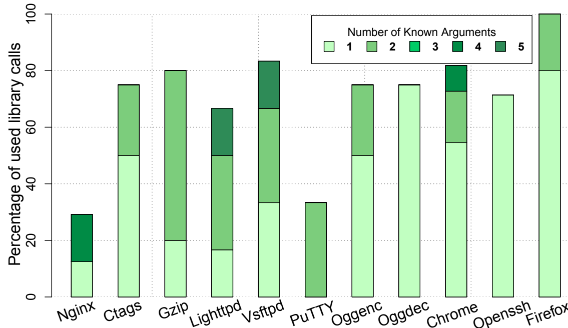

Figure 7: Percentage of critical library functions with at least one known argument across all call sites. The breakdown represents the actual number of known arguments in each application.

that Chrome takes the longest (52 sec). Compared to the overall build time of each of these applications, the time taken by this pass is negligible.

For completeness, we repeated the same analysis when *all* system calls are protected, and present the results in the Appendix. Even for such a worst-case scenario, there is only a modest increase in code and memory size and compilation time, demonstrating the practicality of Saffire in protecting a large number of API functions.

5.5. Application-wide Function Specialization

We begin our analysis of Saffire's static argument binding enforcement by considering application-wide policies (i.e., binding an argument to a set of known values only if this can be applied to all calling contexts of a given function across the application), for the 15 critical functions we have identified. This is in essence the same approach followed by previous works on API specialization, such as Shredder [49].

As shown in Figure 7, in most applications, Saffire can identify arguments with known values across all call sites for around 70% or more of the functions, with the exceptions of Nginx and PuTTY. For example, in Ctags, 50% of the tested library functions have one known argument, while 25% have two known arguments. For Nginx, in contrast, policies can be derived only for two out of the six functions used (33.3%). Although there are a few instances of functions with up to four or five known arguments, in most cases only one or two arguments are known. Overall, these results show that application-wide policies are not very restrictive, and there is potential for much more fine-grained policy enforcement.

5.6. Context-sensitive Function Specialization

In contrast to application-wide policies, Saffire creates context-specific policies that may restrict a different number of arguments across different call sites of the same function. Figure 8 shows the percentage of protected arguments across all call sites for each of the critical functions used by each application (i.e., the Y axis corresponds to each pair of argument and call site, across all call sites of a given function). Light-colored bars correspond to arguments protected with static binding. For example, compared to Figure 7, in which only 33.3% of functions in Nginx have known arguments across all calling contexts, here we see that about 60% of the arguments across all calling contexts can be statically determined (as shown by the leftmost "Overall" bar). Changing the protection mechanism to individual call sites thus provides much better coverage. Next, as described in Section 3.2, we include pointer arguments with known values (medium-colored bars), which provide an additional coverage increase for some functions. For example, the first argument of `mmap64()` in Nginx is always a NULL pointer, which can be statically enforced.

When including dynamic argument binding (dark-colored bars), we observe that the number of protected arguments almost doubles across most applications. Overall, the move from application-wide to per-context policies, and the introduction of dynamic argument binding, achieves complete or nearly complete coverage of all function arguments, depending on the particular combination of application and API function.

To gain more insight on the cases in which Saffire fails to protect some arguments, Table 5 shows the percentage of call sites in which a given argument is protected across all applications. For each function, the second and third

Figure 8: Percentage of protected arguments for the sensitive functions used by each application. Light-colored bars correspond to static argument binding, medium-colored bars to statically enforced pointer arguments, and dark-colored bars to dynamic binding. The leftmost bar of each group corresponds to the average across all functions.

columns correspond to the number of applications that use it and the combined number of its calling contexts across all applications, respectively. The rest of the columns correspond to the individual arguments of each function, and each cell contains the number of calling contexts in which a given argument can be protected by static or dynamic binding. Note that `fstat` and `clone` are missing because they are not used by any of the tested applications. We observe that some pointer arguments (e.g., `addr` and `buf`) are left unprotected, mostly because (as discussed in Section 4.2) they may be passed as arguments to other functions before reaching the final API call, in which case dynamic argument binding cannot be applied. In contrast, arguments corresponding to flags, protection bits, file descriptors, and so on can be fully protected across all call sites most of the time.

5.7. Security Evaluation

We evaluated the effectiveness of Saffire using a set of 17 ROP samples. As discussed earlier, we make the conservative assumption that each of these payloads can be used against each of the tested applications, irrespective of the malicious task they perform. For each combination of ROP payload and application, we identify the library functions used by the payload, and analyze the application's IR to determine if they are also used by the application. For the subset of functions that are used by both, we evaluate how Saffire's context-sensitive policies affect them. In most cases, the combination of library function and argument values fails for at least one of the calls, and thus the attack is prevented. For example, when attackers try to mark a memory region as executable, they use either `mmap()` or `mprotect()`. The arguments to these functions typically never include executable flags in most applications, and thus the calls made by the exploit code are blocked.

The results of this evaluation are presented in the last column of Table 4. We consider an exploit unsuccessful if at least one of the library calls it makes is stopped by static or dynamic argument binding. Saffire was able to break all the ROP payloads in all applications except two (Lighttpd and OpenSSH). Of the 17 ROP payloads used, six use `execve()` to spawn a shell by providing `"/bin/sh"` as an argument. Unfortunately, as shown in Listing 1, Lighttpd includes a call to `fdevent_fork_execve()` with the command to run set as `"/bin/sh"`, which makes this a legitimate control flow transfer within the application—precisely the exact control flow that exploit code also needs. We find similar issues in OpenSSH, in which there is a control path that leads to the execution of `execve("/bin/sh")`.

Listing 1: `execve("/bin/sh")` used in Lighttpd.

```
//src/fdevent.c
static pid_t fdevent_open_logger_pipe_spawn(...) {
    ...
    *(const char **)&args[0] = "/bin/sh";
    ...
    pid = fdevent_fork_execve(args[0], args, ...);
}

pid_t fdevent_fork_execve(const char *name,
        char *argv[], ...) {
    ...
    execve(name, argv, ..);
    ...
}
```

To further investigate Saffire's effectiveness against whole-function reuse attacks [19, 23, 62], we also performed some case studies on Mozilla Firefox, Google Chrome, and Nginx. Given that the full exploit code is not available, our evaluation is performed according to the steps and functions described in the respective papers.

5.7.1. Case Study: Control Jujutsu [23] against Nginx.

Nginx's function `ngx_execute_proc()` contains a call to `execve()`, as shown in Listing 2. We consider the attack scenario in which the indirect call site at `core/ngx_output_chain()` is used to steer the control flow to the target function `ngx_execute_proc()` to spawn a shell. Our backwards analysis though determines that the only possible value for `ctx->path` is the Nginx binary itself, i.e., this call is only used to re-invoke Nginx. Saffire's static argument binding enforces this legitimate value, and thus attacks like Control Jujutsu will fail to reuse it for spawning any other executable.

5.7.2. Case Study: COOP [62] against Firefox.

Given that we did not have access to the actual COOP exploit code, we emulated it by loading Firefox's main library

TABLE 5: Percentage of protected arguments across different calling contexts for all applications.

	Number of Apps	Number of Calling Contexts	Arg1	Arg2	Arg3	Arg4	Arg5	Arg6
mmap(64)	4	10	*addr 10 (100%)	length 10 (100%)	prot 10 (100%)	flags 10 (100%)	fd 10 (100%)	offset 10 (100%)
mprotect	2	11	*addr 7 (64%)	len 11 (100%)	prot 11 (100%)			
open(64)	8	91	*pathname 85 (93.1%)	flags 91 (100%)	mode 91 (100%)			
write	8	242	fd 242 (100%)	*buf 219 (90.5%)	count 242 (100%)			
access	2	8	*pathname 8 (100%)	mode 8 (100%)				
execve	5	13	*filename 13 (100%)	argv[] 13 (100%)	envp[] 13 (100%)			
read	7	60	fd 60 (100%)	*buf 49 (81.6%)	count 56 (100%)			
pread64	1	6	fd 6 (100%)	*buf 6 (100%)	count 6 (100%)	offset 6 (100%)		
fopen	7	53	*pathname 50 (94.3%)	mode 53 (100%)				
fwrite	7	92	*ptr 92 (100%)	size 92 (100%)	nmem 92 (100%)	*stream 91 (98.9%)		
fseek	4	22	*stream 22 (100%)	offset 22 (100%)	whence 22 (100%)			

Listing 2: `execve()` used in Nginx (`ngx_process.c`).

```
static void
ngx_execute_proc(ngx_cycle_t *cycle, void *data) {
  ngx_exec_ctx_t *ctx = data;

  if (execve(ctx->path, ctx->argv, ctx->envp) == -1) {
    ngx_log_error(...);
  }
}
```

`libxul.so` in the address space of a vulnerable application. The gadgets of the exploit attempt to invoke `system("/bin/sh")`. Using Saffire, we analyzed the source code of Firefox v56.0.4 (64-bit) and found that there are five call sites for `system()`, for all of which static argument binding was able to extract the strings used as the command path. This means that Saffire can prevent those call sites from being reused to invoke a shell or other attacker-controlled programs.

5.7.3. Case Study: COOP [19] against Google Chrome. We used Saffire on Chrome v75.0.3732.0 (64-bit Developer Build). As shown in Figure 8, all arguments in most call sites of critical functions can be protected. For example, for `mmap64()`, Saffire restricts the use of memory protection flags, sharing flags, and the offset argument, along with dynamic binding for the file descriptor argument in all of its call sites. We provide a detailed breakdown in Table 6 for three critical functions. For the ML-REC COOP attack [19], the primitive used changes the permissions of a memory region to enable the execution of a second-stage shellcode. Using Saffire, we observe that none of the call sites that change memory protection permissions in Chrome (using either `mprotect()` or `mmap()`) can accept the "execute" permissions, since it is not used by the application. In addition, `execve()` is also restricted similarly to the previous cases.

Table 7 summarizes the advanced whole-function reuse attacks we considered. In all cases, Saffire can prevent whole-function reuse by preventing the use of arbitrary argument values in critical system functions.

TABLE 6: Known arguments for three critical library calls used by whole-function reuse exploits in Google Chrome.

	`mprotect()`	`mmap(64)()`	`execve()`
# Args	3	6	3
# Known Args	1	4	1
Known Arguments	*Memory Protection*	*Memory Protection* *Flag* *File Desc* *Offset*	*Command to run*

TABLE 7: Saffire's effectiveness against whole-function reuse attacks for Firefox, Chrome, and Nginx.

Attack	COOP ML-G [62]	COOP ML-REC [19]	Control Jujutsu [23]
Application	Firefox	Chrome	Nginx
Function	`system()`	`mprotect()`,`mmap()`	`execve()`
Prevented	☑	☑	☑
Mechanism	Arg Value	Arg Value	Arg Value

6. Limitations and Future Work

Saffire is a best-effort, defense-in-depth defense that is not meant to prevent all code reuse attacks. Inspired by previous works on software debloating and specialization, its goal is to hinder the construction of code reuse exploits. As shown from our results, in most cases Saffire protects all arguments for certain calling contexts, but not for all. This means that even if there is one call site with some unprotected argument(s), an attacker may be able to reuse it and invoke the function with a desired argument, as long as a valid control flow path to it can be found. We should note, however, that dynamic argument binding significantly improves coverage, and based on our evaluation, just a few arguments in a few call sites remain unprotected—there are no call sites in which *all* arguments are unprotected. Consequently, if an exploit requires a specific combination

of arguments and none of those values are in the set of known values, the exploit will fail.

Although Saffire falls into the category of software debloating or specialization mitigations, it actually introduces additional code into the protected application, which may seem counter-intuitive in terms of attack surface reduction. However, we believe that expanding the code of the application should not be a concern for multiple reasons. First, as our results demonstrate (Tables 4 and 8), the amount of additional code is negligible, and thus the amount of introduced ROP gadgets due to this code is negligible too. More importantly, the use of CFI precludes these gadgets from being used (except those at the entry points of Saffire's wrapper functions, in which though the attacker is faced with per-context policy checks). In addition, as discussed in Section 5.4, in many cases the wrapper functions of different contexts are identical, and thus can be combined into a single one—we plan to implement this optimization as part of our future work.

To reduce implementation complexity, in our current prototype, only the top-level object is hashed in dynamic argument binding, and any pointers to other nested objects are not followed (the pointers themselves still take part in the hash calculation). This means that attackers can still control the content of any second-level objects pointed to by the parent object—but crucially, not the root object. This is not a significant limitation, as the arguments of the security-critical API functions we are mostly concerned with (see Section 5.1) involve strings, raw memory buffers, and file descriptors, which do not contain pointers to other objects. Still, a recursive hashing scheme (of adjustable depth) could be applied to provide even stronger protection for other APIs that handle complex objects.

Currently, Saffire unconditionally protects all arguments of a given function in a best-effort way. From a security perspective, however, some arguments may not be important, and thus could be left unprotected. Determining which arguments are critical and protecting only those would significantly reduce the extra code and shadow memory space.

Saffire's assumption of a fine-grained CFI mechanism ensures that an attacker is unable to jump over our specialized function wrappers and call API functions directly. This also ensures that the attacker is unable to steer control to an instruction in the middle of the function. The current implementation of Saffire uses LLVM's inbuilt CFI to provide these guarantees. While the CFI mechanism offered by LLVM is not perfect, and under certain circumstances it may still allow for code reuse (e.g., signature-based CFI checks let a function pointer to be assigned to any function with the same signature), for the simplicity of our implementation we relied on this already integrated capability, but other more fine-grained CFI mechanisms can certainly be used [72].

7. Related Work

Non-executable memory [21] and address space randomization [54] have been widely deployed in modern operating systems. However, current exploits rely on code reuse and memory disclosure vulnerabilities to bypass these protections, which has increased the focus of recent research efforts on additional defenses. In the rest of this section, we focus on API-level protections, attack surface reduction, data flow integrity, and advanced code reuse attacks that rely on whole-function reuse—topics that are more closely related to our work.

7.1. Advanced Code Reuse Attacks

Advanced code reuse attacks that can bypass coarse-grained control flow integrity defenses have recently gained popularity. Such attacks follow the same overall approach of ROP in terms of chaining different pieces of code for achieving arbitrary code execution, but differ in the type of these pieces. While ROP gadgets are a few instructions long and end in an indirect branch, advanced attacks try to shape the control flow in accordance to the application's legitimate execution flow.

Counterfeit Object-oriented Programming (COOP) attacks [19, 62] rely on creating a number of counterfeit objects and call small functions in them that perform a specific operation (e.g., read a file to a buffer) using virtual function dispatch. Because these are virtual functions and their call sites cannot be determined statically by control flow analysis, coarse-grained CFI protections cannot prevent such attacks.

Control Jujutsu [23] reuses whole functions from the code of applications like Nginx and Apache that invoke system calls of interest to the attacker. The inputs to these functions are passed to calls like `execve()` that can lead to remote code execution. The example in Listing 2 demonstrates one such code path from Nginx.

7.2. Data Flow Integrity

In 2005, Chen et al. [16] presented non-control-data attacks, which rely on corrupting application data related to authentication or other security-critical operations, which can lead to system compromise. Data flow integrity was introduced by Castro et al. [14] as a means to protect applications against non-control-data attacks by generating a data flow graph. In a more recent work, Kenali [67] performs similar enforcement for the Linux kernel. These and most other works in the area of data flow integrity and isolation [3, 10, 22, 45, 59, 68, 75] aim for whole-application protection and ensure secure data flow across different modules. In contrast, Saffire introduces a lightweight, narrow-scope data integrity enforcement that only ensures a data object is not altered between its creation and its use as an argument to a library call.

7.3. API-level Monitoring

Monitoring execution at the system call or API level strikes a good balance in terms of performance (system call or API invocations are infrequent, e.g., compared to monitoring at the instruction level) and monitoring granularity. As most exploits have to interact with the OS to perform harmful operations, this approach can block this interaction with low performance overhead. Most such protections rely on identifying control flow abnormalities to prevent code reuse attacks. Systems like kBouncer [53] and ROPecker [17] use the Last Branch Record feature of recent processors to inspect the sequence of indirect

branches that lead to sensitive API calls. Similarly, ROP-Guard [27] validates the call precedence of return address and the location of the stack pointer.

Similar approaches have been used in the past by host-based intrusion detection systems. For instance, WHIPS [7], a host-based IDS for Windows 2000, enforces rules kept in an access control database by intercepting native API calls and validating them against the learnt values. Similar systems [4, 48] have used machine learning algorithms to create the policies or rules that are enforced at runtime during the execution of API calls.

Systems following the same approach were prototyped for Linux even earlier. For instance, REMUS [8] implements a reference monitor for system call invocations as a loadable kernel module. Libsafe and Libverify [6] aim to transparently prevent buffer overflows by enforcing buffer sizes and verifying return addresses on the stack through library interposition. Many other systems rely on system call interposition to enforce policies based on allow lists or deny lists [28, 29, 32, 35, 55, 58]. Numerous other works in the span of more than two decades have proposed systems that rely on system call interposition for intrusion detection [9, 24, 25, 26, 39, 44, 63, 74, 76].

7.4. Attack Surface Reduction

Code reuse attacks can rely on any part of code that exists in the address space of a vulnerable process—even code that is not actually used by the application, such as non-imported library functions. Reducing this "attack surface" by removing unused code can thus contribute in hindering the construction of code reuse exploits, although, in most cases, more than enough code still remains to be used by attackers.

As most applications use only a fraction of the functions available in imported libraries, library specialization is a simple and effective code debloating approach which was initially explored for closed-source Windows applications [50]. Piece-wise debloating [57] slims down libraries according to an applications' usage. The information about intra-module dependencies is saved in the ELF binary of the library and at load time, according to what functions are imported, parts of the libraries which are not required are replaced by illegal instructions. Nibbler [2] applies a similar approach at the binary level by extracting the Function Call Graph (FCG) of the binary and all imported libraries to create an application-level FCG and remove any unreachable code. LibFilter [65] is a system that identifies unused functions in an application's dynamically-linked libraries. Confine [30] statically identifies the set of system calls used in a container deployment and restricts access to only those needed. As already discussed, in our previous work we proposed Shredder [49], which moves one step further and restricts the input argument values of the remaining functions, after any of the above code specialization approaches has been applied.

Software winnowing [46] specializes both application and library code. The authors have implemented a code specialization tool on top of LLVM, called OCCAM (Object Culling and Concretization for Assurance Maximization). OCCAM generates specialized versions of applications according to the build time configuration and deployment context. BinTrimmer [60] uses abstract interpretation to generate a near-precise control flow graph from a binary to identify unused basic blocks, and rewrites them with useless instructions. Razor [56] debloats deployed binaries by collecting sample test cases from users, and augmenting them with control-flow heuristics to infer code that is necessary to support user-expected functionalities.

Configuration-driven debloating [36] identifies parts of an application's code which are not used under specific runtime configurations. Code debloating has also been explored in other domains, such as PHP applications [5] and the kernel [38, 40, 41, 42, 47, 70, 79].

8. Conclusion

We presented Saffire, a compiler-level defense that performs context-sensitive function customization against code reuse attacks. Saffire transforms each call site of a critical function to invoke a custom function that applies i) static argument binding, to eliminate arguments with static values and concretize them within the function body, and ii) dynamic argument binding, to confine the values of arguments that cannot be statically derived. We have demonstrated the effectiveness and practicality of Saffire in preventing real-world exploits. Besides its more comprehensive protection compared to previous API specialization approaches [49], which includes non-static arguments, another important key benefit is its effectiveness against whole-function reuse attacks, which currently pose a challenge to CFI defenses. As a best-effort, defense-in-depth approach, we believe that Saffire is a practical solution that complements existing CFI and library specialization approaches with a unique set of additional protection capabilities.

Acknowledgments

This work was supported by the National Science Foundation (NSF) through awards CNS-1749895 and CNS-1617902, the Office of Naval Research (ONR) through award N00014-17-1-2891, and the Defense Advanced Research Projects Agency (DARPA) through award D18AP00045. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF, ONR, or DARPA.

References

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS)*, 2005, pp. 340–353.

[2] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating binary shared libraries," in *Proceedings of the 35th Annual Computer Software and Applications Conference (ACSAC)*, 2019.

[3] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 2008, pp. 263–277.

[4] M. Alazab, S. Venkatraman, P. Watters, and M. Alazab, "Zero-day malware detection based on supervised learning algorithms of API call signatures," in *Proceedings of the 9th Australasian Data Mining Conference (AusDM)*, 2011, pp. 171–182.

[5] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: quantifying the security benefits of debloating web applications," in *Proceedings of the 28th USENIX Security Symposium*, 2019.

[6] A. Baratloo, N. Singh, and T. Tsai, “Transparent run-time defense against stack smashing attacks,” in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2000.

[7] R. Battistoni, E. Gabrielli, and L. V. Mancini, “A host intrusion prevention system for Windows operating systems,” in *Proceedings of the 9th European Symposium on Research in Computer Security (ESORICS)*, 2004, pp. 352–368.

[8] M. Bernaschi, E. Gabrielli, and L. V. Mancini, “Remus: A security-enhanced operating system,” *ACM Trans. Inf. Syst. Secur.*, vol. 5, no. 1, pp. 36–61, Feb. 2002.

[9] S. Bhatkar, A. Chaturvedi, and R. Sekar, “Dataflow anomaly detection,” in *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 2006.

[10] P. Biswas, A. Di Federico, S. A. Carr, P. Rajasekaran, S. Volckaert, Y. Na, M. Franz, and M. Payer, “Venerable variadic vulnerabilities vanquished,” in *Proceedings of the 26th USENIX Security Symposium*, 2017, pp. 183–198.

[11] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, “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, pp. 30–40.

[12] E. Bosman and H. Bos, “Framing signals - a return to portable shellcode,” in *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 2014, pp. 243–258.

[13] N. Carlini, A. Barresi, M. Payer, D. Wagner, and T. R. Gross, “Control-flow bending: On the effectiveness of control-flow integrity,” in *Proceedings of the 24th USENIX Security Symposium*, 2015.

[14] M. Castro, M. Costa, and T. Harris, “Securing software by enforcing data-flow integrity,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[15] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy, “Return-oriented programming without returns,” in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010, pp. 559–572.

[16] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, “Non-control-data attacks are realistic threats,” in *Proceedings of the 14th USENIX Security Symposium*, 2005.

[17] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, “ROPecker: A generic and practical approach for defending against ROP attacks,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[18] J. Corbet, “Memory protection keys,” https://lwn.net/Articles/643797/, 2015.

[19] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, “It’s a TRaP: Table randomization and protection against function-reuse attacks,” in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 243–255.

[20] S. Designer, “Getting around non-executable stack (and fix),” 1997, http://seclists.org/bugtraq/1997/Aug/63.

[21] ——, “Non-executable stack patch,” http://lkml.iu.edu/hypermail/linux/kernel/9706.0/0341.html, 1997.

[22] U. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, “XFI: Software guards for system address spaces,” in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, 2006, pp. 75–88.

[23] I. Evans, F. Long, U. Otgonbaatar, H. Shrobe, M. Rinard, H. Okhravi, and S. Sidiroglou-Douskos, “Control jujutsu: On the weaknesses of fine-grained control flow integrity,” in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 901–913.

[24] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, “Anomaly detection using call stack information,” in *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 2003, pp. 62–75.

[25] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller, “Formalizing sensitivity in static analysis for intrusion detection,” pp. 194–208, 2004.

[26] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, “A sense of self for Unix processes,” in *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 1996.

[27] I. Fratrić, “ROPGuard: Runtime prevention of return-oriented programming attacks,” 2012, http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf.

[28] T. Garfinkel, “Traps and pitfalls: Practical problems in system call interposition based security tools,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2003.

[29] T. Garfinkel, B. Pfaff, and M. Rosenblum, “Ostia: A delegating architecture for secure system call interposition,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2003.

[30] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, “Confine: Automated system call policy generation for container attack surface reduction,” in *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

[31] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis, “Out of control: Overcoming control-flow integrity,” in *Proceedings of the 35th IEEE Symposium on Security & Privacy (S&P)*, 2014.

[32] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, “A secure environment for untrusted helper applications (confining the wily hacker),” in *Proceedings of the 6th USENIX Security Symposium*, 1996.

[33] A. G. Hashim Sharif, Muhammad Abubakar and F. Zaffar, “Trimmer: Application specialization for code debloating,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

[34] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, “Effective program debloating via reinforcement learning,” in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2018.

[35] K. Jain and R. Sekar, “User-level infrastructure for system call interposition: A platform for intrusion detection and confinement,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2000.

[36] H. Koo, S. Ghavamnia, and M. Polychronakis, “Configuration-driven software debloating,” in *Proceedings of the 12th European Workshop on System Security (EuroSec)*, 2019.

[37] S. Krahmer, “x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique,” 2005, http://www.suse.de/~krahmer/no-nx.pdf.

[38] T. Kroes, A. Altinay, J. Nash, Y. Na, S. Volckaert, H. Bos, M. Franz, and C. Giuffrida, “BinRec: Attack surface reduction through dynamic binary recovery,” in *Proceedings of the 2018 Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2018.

[39] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, “On the detection of anomalous system call arguments,” in *Proceedings of the 8th European Symposium on Research in Computer Security (ESORICS)*, 2003, pp. 326–343.

[40] A. Kurmus, S. Dechand, and R. Kapitza, “Quantifiable run-time kernel attack surface reduction,” in *Proceedings of the 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2014, pp. 212–234.

[41] A. Kurmus, A. Sorniotti, and R. Kapitza, “Attack surface reduction for commodity OS kernels: Trimmed garden plants may attract less bugs,” in *Proceedings of the 4th European Workshop on System Security (EuroSec)*, 2011.

[42] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza, “Attack surface metrics and automated compile-time OS kernel tailoring,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.

[43] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity,” in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014, pp. 147–163.

[44] P. Li, H. Park, D. Gao, and J. Fu, “Bridging the gap between data-flow and control-flow analysis for anomaly detection,” in *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2008, pp. 392–401.

[45] T. Liu, G. Shi, L. Chen, F. Zhang, Y. Yang, and J. Zhang, “TMDFI: Tagged memory assisted for fine-grained data-flow integrity towards embedded systems against software exploitation,” in *Proceedings of the 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications (TrustCom)*, 2018, pp. 545–550.

[46] G. Malecha, A. Gehani, and N. Shankar, “Automated software winnowing,” in *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC)*, 2015, pp. 1504–1511.

[47] H. M. Mansour Alharthi, Hong Hu and T. Kim, “On the effectiveness of kernel debloating via compile-time configuration,” in *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2018.

[48] Miao Wang, Cheng Zhang, and Jingjing Yu, “Native API based Windows anomaly intrusion detection method using SVM,” in *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC)*, 2006.

[49] S. Mishra and M. Polychronakis, “Shredder: Breaking exploits through API specialization,” in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

[50] C. Mulliner and M. Neugschwandtner, “Breaking payloads with runtime code stripping and image freezing.” Black Hat USA, 2015.

[51] Nergal, “The advanced return-into-lib(c) exploits: PaX case study,” *Phrack*, vol. 11, no. 58, Dec. 2001.

[52] T. Newsham, “Non-exec stack,” 2000, http://seclists.org/bugtraq/2000/May/90.

[53] V. Pappas, M. Polychronakis, and A. D. Keromytis, “Transparent ROP exploit mitigation using indirect branch tracing,” in *Proceedings of the 22nd USENIX Security Symposium*, August 2013.

[54] PaX Team, “Address space layout randomization,” 2003, http://pax.grsecurity.net/docs/aslr.txt.

[55] M. Payer and T. R. Gross, “Fine-grained user-space security through virtualization,” in *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2011, pp. 157–168.

[56] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, “RAZOR: A framework for post-deployment software debloating,” in *Proceedings of the 28th USENIX Security Symposium*, 2019.

[57] A. Quach, A. Prakash, and L. K. Yan, “Debloating software through piece-wise compilation and loading,” in *Proceedings of the 27th USENIX Security Symposium*, 2018.

[58] M. Rajagopalan, M. Hiltunen, T. Jim, and R. Schlichting, “Authenticated system calls,” in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2005, pp. 358–367.

[59] T. Ramezanifarkhani and M. Razzazi, “Principles of data flow integrity: Specification and enforcement.” *J. Inf. Sci. Eng.*, vol. 31, no. 2, pp. 529–546, 2015.

[60] N. Redini, R. Wang, A. Machiry, Y. Shoshitaishvili, G. Vigna, and C. Kruegel, “BinTrimmer: Towards static binary debloating through abstract interpretation,” in *Proceedings of the 16th International Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2019, pp. 482–501.

[61] R. Rudd, R. Skowyra, D. Bigelow, V. Dedhia, T. Hobson, C. L. Stephen Crane, P. Larsen, L. Davi, M. Franz, A.-R. Sadeghi, and H. Okhravi, “Address-Oblivious Code Reuse: On the Effectiveness of Leakage Resilient Diversity,” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[62] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, “Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications,” in *Proceedings of the 36th IEEE Symposium on Security & Privacy (S&P)*, 2015, pp. 745–762.

[63] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, “A fast automaton-based method for detecting anomalous program behaviors,” in *Proceedings of the IEEE Symposium on Security & Privacy*, 2001, pp. 144–155.

[64] H. Shacham, “The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86),” in *Proceedings of the 14th ACM conference on Computer and Communications Security (CCS)*, 2007, pp. 552–561.

[65] B. Shteinfeld, “Libfilter: Debloating dynamically-linked libraries through binary recompilation,” Undergraduate Honors Thesis, Brown University, 2019.

[66] K. Z. Snow, L. Davi, A. Dmitrienko, C. Liebchen, F. Monrose, and A.-R. Sadeghi, “Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization,” in *Proceedings of the 34th IEEE Symposium on Security & Privacy (S&P)*, 2013.

[67] C. Song, B. Lee, K. Lu, W. Harris, T. Kim, and W. Lee, “Enforcing kernel security invariants with data flow integrity.” in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2016.

[68] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek, “HDFI: Hardware-assisted data-flow isolation,” in *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, 2016.

[69] L. Song and X. Xing, “Fine-grained library customization,” in *Proceedings of the ECOOP 1st International Workshop on SoftwAre debLoating And Delayering (SALAD)*, 2018.

[70] R. Tartler, A. Kurmus, B. Heinloth, V. Rothberg, A. Ruprecht, D. Dorneanu, R. Kapitza, W. Schröder-Preikschat, and D. Lohmann, “Automatic OS kernel TCB reduction by leveraging compile-time configurability,” in *Proceedings of the 8th USENIX Conference on Hot Topics in System Dependability (HotDep)*, 2012.

[71] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, “ERIM: Secure, efficient in-process isolation with protection keys (MPK),” in *Proceedings of the 28th USENIX Security Symposium*, 2019.

[72] V. van der Veen, D. Andriesse, E. Göktaundefined, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, “Practical context-sensitive CFI,” in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, 2015.

[73] V. van der Veen, D. Andriesse, M. Stamatogiannakis, X. Chen, H. Bos, and C. Giuffrdia, “The dynamics of innocent flesh on the bone: Code reuse ten years later,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017, pp. 1675–1689.

[74] D. Wagner and R. Dean, “Intrusion detection via static analysis,” in *Proceedings of the IEEE Symposium on Security & Privacy*, 2001.

[75] W. Wang, X. Xu, and K. W. Hamlen, “Object flow integrity,” in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017, pp. 1909–1924.

[76] A. Wespi, M. Dacier, and H. Debar, “Intrusion detection using variable-length audit trail patterns,” in *Proceedings of the 3rd Conference in Recent Advances in Intrusion Detection (RAID)*, 2000, pp. 110–129.

[77] D. W. Yufei Jiang and P. Liu, “Jred: Program customization and bloatware mitigation based on static analysis,” in *Proceedings of the 40th Annual Computer Software and Applications Conference (ACSAC)*, 2016.

[78] T. L. Yurong Chen and G. Venkataramani, “Damgate: Dynamic adaptive multi-feature gating in program binaries,” in *Proceedings of the Second Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, 2017.

[79] X. Z. Zhongshu Gu, Brendan Saltaformaggio and D. Xu, “Face-change: Application-driven dynamic kernel view switching in a virtual machine,” in *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

TABLE 8: Code and memory increase, time for Saffire's LLVM pass, and number of ROP payloads blocked (out of 17) when *all* libc functions are protected.

Application	# of Functions Added	Size of Shadow Memory per Thread (8-byte units)	# of IR Statements Added	Time Taken to Run Pass (sec)	# of ROP Payloads Blocked
Nginx	288 (15%)	454	7117 (1.1%)	42.4	17
Ctags	39 (3%)	49	1147 (0.4%)	9.3	17
Gzip	188 (153%)	345	2142 (19.6%)	11	17
Lighttpd	285 (45%)	392	2611 (7.3%)	14.1	12
Vsftpd	147 (58%)	149	784 (7.8%)	6.8	17
PuTTY	183 (14%)	234	2102 (1.5%)	10.2	17
OggEnc	42 (39%)	87	1327 (28.8%)	3.4	17
OggDec	16 (40%)	36	628 (56.3%)	3.2	17
Chrome	277 (3%)	383	3132 (0.2%)	71.1	17
Openssh	296 (51%)	401	6709 (4.2%)	30.8	12
Firefox	79 (18%)	65	1372 (0.8%)	18.2	17

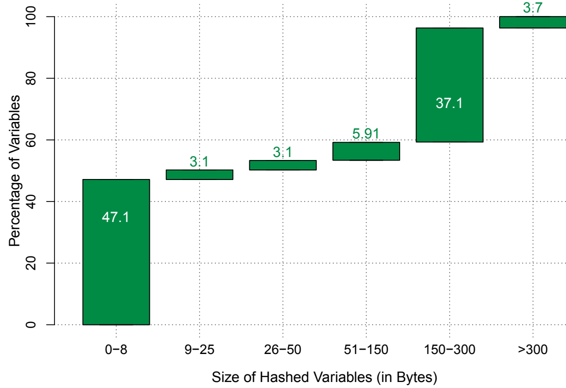

Figure 9: Distribution of argument sizes for dynamic binding. About half of the arguments are eight bytes or smaller, while arguments in the range of 150–300 bytes are mostly FILE structs, each 216 bytes in size.

Appendix

1. Performance Evaluation with All System Calls Instrumented

To gain further insight regarding Saffire's worst-case performance, we repeated the measurements of code and memory overhead discussed in Section 5.4 when protecting *all* system calls. We identified 322 system calls, of which 18 do not take any input arguments and are thus left out, and performed the same analysis as presented in Table 4.

Table 8 presents this new set of results. The process of instrumenting all system calls inevitably increases the values in all columns of the table (except the number of blocked ROP payloads, which, as expected, does not change). As we see in the second column, for Gzip, the number of new functions added to the binary is 1.5 times more than the original. The reason is the same as explained in Section 5.4, which is that there are more than a hundred instances of calls to `write()`. In the fourth column, where we mention the increase in the number of IR statements, we see that for OggDec, the increase is about 56%, while for Chrome, it is 0.2%. Hence, we conclude that for large applications like Chrome and Nginx, using Saffire with all system calls does not impact significantly the size of the binaries or the number of instructions added.

TABLE 9: Linux ROP payloads used in our evaluation.

1) Return Oriented Programming and ROPgadget tool
`http://shell-storm.org/blog/Return-Oriented-Programming-and-ROPgadget-tool/`
2) ARM Exploitation - Defeating DEP - executing mprotect()
`https://blog.3or.de/arm-exploitation-defeating-dep-executing-mprotect.html`
3) 64-bit ROP — You rule 'em all!
`https://0x00sec.org/t/64-bit-rop-you-rule-em-all/1937`
4) 64-bit Linux Return-Oriented Programming
`https://crypto.stanford.edu/~blynn/rop/`
5) Return-Oriented-Programming(ROP FTW)
`http://www.exploit-db.com/docs/english/28479-return-oriented-programming-(rop-ftw).pdf`
6) PMS 0.42 - Local Stack-Based Overflow (ROP)
`https://www.exploit-db.com/exploits/44426/`
7) Crashmail 1.6 - Stack-Based Buffer Overflow (ROP)
`https://www.exploit-db.com/exploits/44331/`
8) PHP 5.3.6 - Local Buffer Overflow (ROP)
`https://www.exploit-db.com/exploits/17486/`
9) HT Editor 2.0.20 - Local Buffer Overflow (ROP)
`https://www.exploit-db.com/exploits/22683/`
10) Bypassing non-executable memory, ASLR and stack canaries on x86-64 Linux
`https://www.antoniobarresi.com/security/exploitdev/2014/05/03/64bitexploitation/`
11) Bypassing non-executable-stack during Exploitation (return-to-libc)
`https://www.exploit-db.com/papers/13204/`
12) Exploitation - Returning into libc
`https://www.exploit-db.com/papers/13197/`
13) Bypass DEP/NX and ASLR with Return Oriented Programming technique
`https://medium.com/4ndr3w/linux-x86-bypass-dep-nx-and-aslr-with-return-oriented-programming-ef4768363c9a/`
14) ROP-CTF101
`https://ctf101.org/binary-exploitation/return-oriented-programming/`
15) Introduction to return oriented programming (ROP)
`https://codearcana.com/posts/2013/05/28/introduction-to-return-oriented-programming-rop.html/`
16) Simple ROP Exploit Example
`https://gist.github.com/mayanez/c6bb9f2a26fa75261a9a26a0a637531b/`
17) Analysis of Defenses against Return Oriented Programming
`https://www.eit.lth.se/sprapport.php?uid=829/`

2. Argument Size Distribution

Dynamic argument binding involves computing the HMAC of arguments, which can potentially result in processing large amounts of data for long strings or large objects. To gain more insight about the amount of data that needs to be processed, we performed an experiment to measure the sizes of the arguments protected by dynamic binding during a sample run of the tested applications. From Figure 9, we see that about 47% of the hashed arguments have a size of eight bytes or smaller. The other major size bucket (37%) corresponds to sizes of 150–300 bytes, which after some investigation we can attribute to FILE structures, the size of which is 216 bytes. Overall, dynamic argument binding treats only small objects, and thus computing their HMAC does not incur any significant runtime overhead.