There is Safety in Numbers: Preventing Control-Flow Hijacking by Duplication

Job Noorman, Nick Nikiforakis, and Frank Piessens

IBBT-DistriNet, KU Leuven
job.noorman@student.kuleuven.be
{nick.nikiforakis,frank.piessens}@cs.kuleuven.be

Abstract. Despite the large number of proposed countermeasures against control-flow hijacking attacks, these attacks still pose a great threat for today's applications. The problem with existing solutions is that they either provide incomplete probabilistic protection (e.g., stack canaries) or impose a high runtime overhead (e.g., bounds checking).
In this paper, we show how the concept of program-part duplication can be used to protect against control-flow hijacking attacks and present two different instantiations of the duplication concept which protect against popular attack vectors. First, we use the duplication of functions to eliminate the need of return addresses and thus provide complete protection against attacks targeting a function's return address. Then we demonstrate how the integrity of function pointers can be protected through the use of data duplication. We test the combined effectiveness of our two methods and experimentally show that they provide an almost complete protection against control-flow hijacking attacks with only a low runtime overhead in real-world applications.

Keywords: control-data attacks, duplication, return addresses, function pointers

1 Introduction

All but the simplest of programs contain non-straight-line code, i.e. code for which the control-flow is not fixed at compile time. The control-flow of those programs will be dictated by data structures located somewhere in memory. These control structures normally take the form of a memory address where the next instruction to be executed is located; the most common examples are return addresses and function pointers. If the program contains some kind of vulnerability that gives the attacker the opportunity to write arbitrary data to a control structure, he will be able to hijack the control-flow.

The problem of control-flow hijacking has been known for a long time; the Morris worm [28] in 1988 was the first popular attack to exploit a stack-based buffer overflow to this end. As such, this problem has gained much attention from the academic world and numerous solutions have been proposed. The most widely

adopted solution for protecting return addresses is the use of stack canaries (e.g. StackGuard [9] for GCC). While very effective, stack canaries are no silver bullet for the problem of return address smashing. The reason for this is threefold. Firstly, stack canaries can only detect return address smashing through a buffer overflow. If the attacker manages to overwrite the return address through an indirect pointer overwrite [6], he will be able to bypass the protection mechanism. Secondly, the effectiveness of stack canaries relies on a secret value that is located somewhere in memory. If the attacker somehow gains access to this memory location by either overwriting it or disclosing its contents, the protection becomes useless. For instance, prior research has shown that buffer over-reads and format-string vulnerabilities can be used to uncover secrets hidden on the stack or heap of a protected application [31]. Lastly, most implementations will not protect every function by default but rather use a heuristic to decide which functions to protect. This heuristic usually takes the form of only protecting functions that have a stack-allocated character buffer of a certain minimum size and is used to minimize the performance overhead of stack canaries. A recent exploit of `LibTIFF` [24] used an overflow in an integer array and, as such, was not detectable by stack canaries.

A second attack vector that is used to hijack the control flow of a program, are function pointers. The protection of function pointers, however, has gained much less attention than the protection of return addresses, probably because of their relatively infrequent use in applications. One proposed technique to protect function pointers is the use of encryption by PointGuard [8]. Here, pointer values are encrypted (using XOR with a secret value) when stored and decrypted when used. This technique, like stack canaries, relies on a secret value that the attacker cannot know for the protection to remain effective. Again, it has been shown that this protection may be circumvented in the presence of a buffer over-read vulnerability [31].

In this paper, two techniques are introduced that use duplication to prevent control-flow hijacking. The first technique is a novel compile-time solution of protecting return addresses. Our system duplicates functions in such a way that makes return addresses unnecessary for the program to run correctly. Since instead of trying to *protect* return addresses we completely *remove* them from the program, there simply is no return control-structure left to be attacked. Therefore, this technique provides a more thorough solution to the problem of return address smashing than any popular countermeasure that tries to protect return addresses, including stack canaries. As an added benefit, this technique proves to have little or no overhead since there are no runtime checks to be performed. In fact, programs protected by this scheme in some cases outperform their unprotected versions.

The second technique this paper introduces uses the duplication of control data to protect function pointers. Function pointers used by a program are duplicated in a protected storage so that they can be checked for corruption when used. This technique is similar to that of the Return Address Defender [7], but applied to function pointers instead of return addresses. Because of the infrequent

use of function pointers (as opposed to return addresses), the runtime overhead of our technique has proven to be small for most programs.

The rest of this paper is structured as follows: Section 2 gives the general principles of how duplication can be used to prevent control-flow hijacking and shows the design of two different instantiations of this principle. Section 3 gives a short overview of how these instantiations were implemented. In Section 4, we evaluate the two introduced techniques in terms of effectiveness in protecting programs and their overhead on the runtime performance and program size, followed by a discussion of some issues we encountered in Section 5. Section 6 discusses related work and we conclude in Section 7.

2 Design

2.1 Using Duplication for Protection

There are two ways in which duplication can be used to protect control data in memory: (a) duplicate the control data itself so that it becomes more difficult to overwrite all copies, or (b) duplicate some other part of the program so that the control data is not needed any more. Using (a) is highly generic: any kind of data can be protected in this way. How well this protection works, however, depends on how the duplicated control data is protected since an attacker might be able to find a way to overwrite both copies and circumvent the protection. Also, this technique will always cause runtime overhead on the protected program.

Technique (b), on the other hand, offers full protection of the control data by completely removing it. Another asset of this technique is that it does not rely on any runtime checks so it might be possible to implement it without any runtime overhead on the protected program. Unfortunately, it is not possible to protect all types of control data by this scheme. The next sections explain instantiations of both schemes.

2.2 Protecting Return Addresses

Idea. The key idea of our approach is that return addresses are only needed if a function is called from more than one place. Indeed, if a function has only one call site, the return instruction can be replaced by a jump instruction to a hard coded label just after the call site. Figure 1 illustrates this. The transformed function always semantically behaves the same as the original as long as it is only called from one call site.

The whole purpose of having functions, however, is to enable code-reuse. Therefore, most programs will likely not have many functions that are only called from a single call site. This is where function duplication comes in: if we make as many copies of a function as it has call sites, all return instructions in those copies can be eliminated.

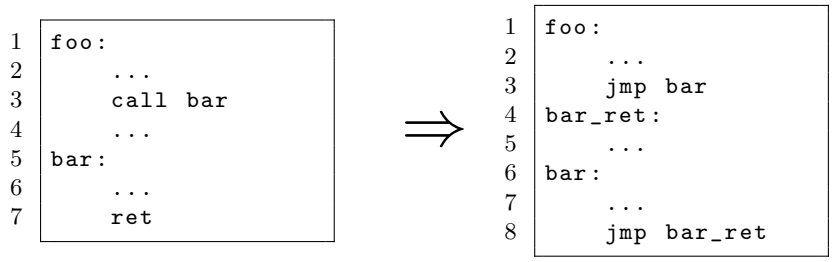


Fig. 1. Transforming a return instruction into a jump

Duplicating Functions. To duplicate functions in such a way that every function has only one call site, information is needed about which function calls are made in a program. More specifically, we need to know for each function by which functions it is called. This information is available in the call graph of the program. A call graph is a directed graph in which every function of the program is represented by a node and there is an edge from node A to node B if function A calls function B. For example, the call graph of the program in Fig. 2 is shown in Fig. 3a.

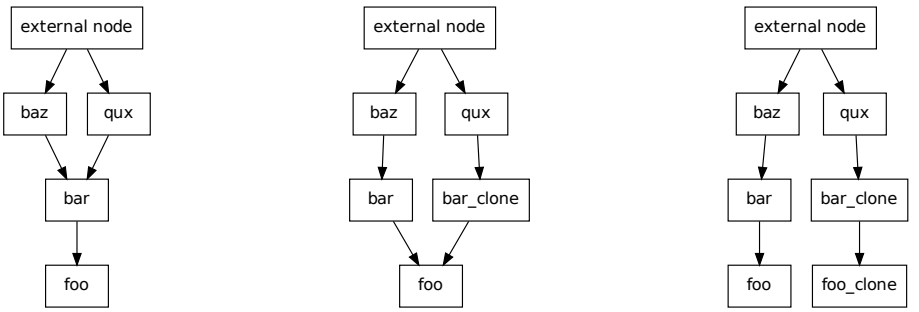
```
1 static void foo() {}
2 static void bar() {foo();}
3 void baz() {bar();}
4 void qux() {bar();}
```

Fig. 2. An example C program

There is one special node visible in this call graph: the external node. A node has an incoming edge from the external node if it might be called from an unknown function. In this example, the functions `baz` and `qux` have external linkage (i.e., not declared with the `static` keyword in C) which means they might be called from functions not available at the time of compilation. This is important information since it means that we cannot eliminate return instructions in functions that are called from the external node. Indeed, it is impossible to compute, at the time of compilation, where these functions should return to.

Through the use of the information available in the call graph, the process of duplicating functions is straightforward. Starting from the external node, every node is visited once. If a node is visited that has more than one incoming edge, it is copied as many times as there are incoming edges and the call graph is updated accordingly. For the example C program shown in Fig. 2, the nodes corresponding to the function `baz` and `qux` are visited first. Since both have only one incoming edge, nothing needs to be done. Next, the node for the function `bar`

is visited, which has two incoming edges. Duplicating this function introduces a new function and the original call sites are updated. This is illustrated in Fig. 3b. Note that due to this transformation, the function `foo`, which originally had only one call site, now has two. The node for `bar_clone`, which has only one incoming edge, is visited next. Then `foo` is visited which needs to be duplicated. When this is done, the process is complete. The resulting call graph is shown in Fig. 3c.



(a) Call graph of the example program.

(b) Call graph after `bar` has been duplicated.

(c) Call graph after the duplication process.

Fig. 3. Example call graph transformation.

At this point, the program has been transformed to a version that is semantically equivalent to the original but has only functions that are called by at most one other function. This means the principle explained in the previous section can be applied in a straightforward way. First, we add a label after all call instructions, which will allow the called function to jump back to this point. Then, in the called function all return instructions are replaced by direct jumps to the inserted label.

Recursive Functions. The aforementioned technique of function duplication does not work for recursive functions, since recursive functions cannot be duplicated in a way that makes them have only one call site. The same property holds for groups of mutually recursive functions. As a result, we cannot know at compile time where a recursive function will return to. To solve this problem, this decision has to be postponed to runtime.

The approach taken in our solution is the following: first, the functions in the call graph are grouped in sets of mutually recursive functions; these sets are the strongly connected components of the call graph[1]. Then duplication is performed as usual but instead of duplicating functions, these sets are duplicated. The result

[1] A strongly connected component of a directed graph is a maximally sized subgraph in which each node can reach all other nodes

of this step is a call graph in which each mutually recursive group of functions has only one call site from outside this group. This assures that each function has the lowest possible number of call sites.

When eliminating return instructions, each call to a function having more than one call site will push a different return index on the stack. When the called function needs to return, it will pop this index and use it to decide where to return to. If an invalid index is encountered when returning, the program is aborted. Note that this approach is similar is the one taken by Li et al. in [15]. However, they make use of global return indices (i.e., every call site in the program has a unique return index) whereas in our approach, these indices are local to functions. Also, since return indices possibly introduce a new attack vector, only (mutually) recursive functions make use of these indices; normal functions always jump to the same location when returning. We discuss this in more detail in Section 5.

Function Duplication versus Function Inlining Another way that return addresses could be eliminated from a program is through function inlining, a process where the compiler incorporates functions in the body of their callers. While this is a viable approach for simple functions, it cannot be used for (mutually) recursive functions. At the same time, the incorporation of functions that are not frequently used in the body of functions that are, will likely lead to increased cache misses which will in turn incur a higher performance overhead. For these two reasons, we decided that the maintenance of functions as stand-alone chunks of code with hard-coded return addresses is the best of the two approaches and thus favored it over function inlining.

2.3 Protecting Function Pointers

For the protection of function pointers, we have taken the approach of duplicating the pointers themselves. The source code of a program is automatically transformed so that when a value is stored in a function pointer, it is duplicated in another part of memory and when it is loaded, the loaded value is compared with the duplicated value. If these values do not match, the program is aborted.

Although duplication alone would provide some protection – the attacker would need to find a way to overwrite two distinct memory locations with the same value – our technique additionally protects the memory locations where duplicates are stored. Two different techniques for protecting these duplicates are provided by our solution: (a) the use of unwritable guard pages around the storage, or (b) making the entire storage location unwritable while it is not needed. While (b) is clearly the most secure of the two, it also incurs the most overhead since two calls to `mprotect` (thus, two system calls) are needed whenever a pointer is stored: one to unlock the storage and one to lock it again. Note that no system calls are needed when a pointer is loaded since the storage remains readable at all times.

3 Implementation Details

The techniques presented in this paper have been implemented using the LLVM Compiler Infrastructure [17]. Compilation of a program using LLVM is a three-step process: (a) a frontend translates the source language in the LLVM Intermediate Representation (IR), (b) architecture independent optimizations are run on the IR, and (c) a backend translates the IR to target instructions and runs architecture dependent optimizations.

3.1 Protecting Return Addresses

The technique of duplicating functions to protect return addresses has been implemented as two passes in LLVM. The first pass is a transformation of the IR to bring the program in the form discussed in Section 2.2: every function has only one call site. Because it transforms the IR, it is completely independent of the source language and the target architecture. Unfortunately, the IR has some restrictions which make it impossible to eliminate return instructions at this point. More specifically, the IR does not allow jumping outside of functions, which is exactly what is needed to replace return instruction by jumps.

The solution to this problem is to eliminate return instructions in the backend where it is possible to insert arbitrary instructions of the target architecture. The downside of this approach is that the implementation is not architecture independent any more. Currently, support has been implemented for x86 (32 and 64 bit), which is the platform of choice for most desktop and server environments.

3.2 Protecting Function Pointers

The implementation of our protection of function pointers contains two parts: a secure storage mechanism and a program transformation that inserts calls to this storage. The secure storage is implemented as a fixed-size hash map to provide fast insertions and lookups of pointers. The hash map maps the address of a memory location to the value of the function pointer at that memory location. The size of the storage, as well as the desired protection mechanism, can be configured through environment variables.

The transformation of the program is implemented as a pass in LLVM. It is a transformation of the LLVM IR so it is completely independent of the source language or target architecture. The transformation itself is straightforward: all loads and stores of function pointers are replaced with calls to the secured storage. This is illustrated in Fig. 4. There are three transformations visible in this figure: (a) assignments to a function pointer are replaced by a call to `__store_ptr()`. This function stores the pointer in the protected storage as well as in the original location. (b) Calls of a function pointer are replaced by a call to `__load_ptr()` which loads the pointer from the storage and checks for equality with the value at the original location. The pointer is loaded in a register and then this register is used to call the function. (c) If the lifetime of the protected pointer ends, its storage space is released with `__free_ptr()`. Currently, this is only implemented for stack based function pointers.

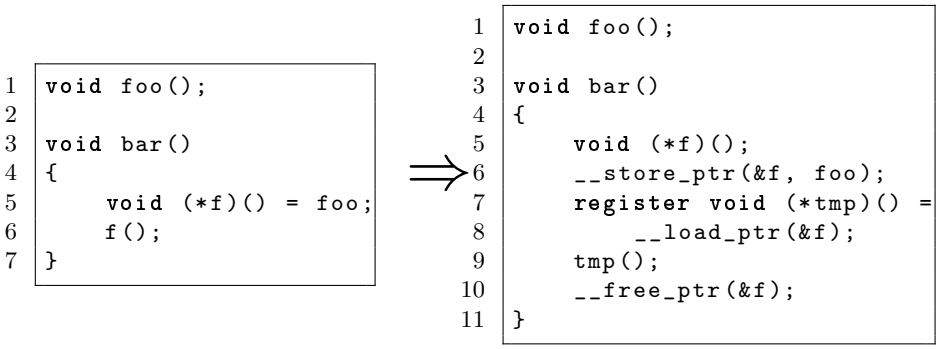


Fig. 4. Transformation used for pointer duplication

4 Evaluation

4.1 Security evaluation

We evaluated the provided protection of our solutions using RIPE [36], an open source testbed for quantifying the protection of any given countermeasure. RIPE performs a plethora of different attacks on itself and reports each attack's success or failure. For example, it is able to perform direct (e.g., buffer overflows) and indirect attacks on return addresses, function pointers and `longjmp` buffers[2] located in all memory segments (e.g., stack, heap and BSS). All experiments were performed on a Linux system configured to not use any protection mechanisms.

The results of RIPE are shown in Table 1. For the protection of return addresses, it is clear that our proposed technique of duplicating functions is more effective than stack canaries[3]. This is because, as explained in Section 1, stack canaries fail to protect against indirect pointer overwrite attacks while our technique does protect against such attacks.

The table also shows that using the pointer duplication technique is very effective: on its own, it protects against more than half of the attacks that were possible in the unprotected version of RIPE. The reason that the protection of function pointers seems to be more effective than that of return addresses is that RIPE is able to perform more attacks on the former. When combining our two techniques, the only attacks that still succeed are attacks abusing `longjmp` buffers. Given the fact that newer versions of `libc` actively protect this type of structures, the same LLVM-compiled and protected binary produces zero successful attacks on a more modern system.

[2] Longjmp buffers are used to store the program state (e.g. program counter and stack pointer) between calls of `setjmp()` and `longjmp()` and are a popular attack vector.
[3] We use the stack canary implementation of LLVM which is similar to StackGuard

Table 1. The number of successful attacks in RIPE when using different protections.

Protection type	Successful attacks
No protection	540
Stack canaries (1)	520
Duplicated functions (2)	470
Duplicated function pointers (3)	230
(1) + (3)	210
(2) + (3)	180

4.2 Performance evaluation

In our evaluation, we compare the runtime performance of applications protected by our techniques to that of unprotected applications. Duplicated function pointers are protected by guard pages to give a baseline of the overhead incurred by this countermeasure. We also show the performance of each application when protected by stack canaries since our proposed technique of duplicating functions is an alternative to that approach.

Two types of performance benchmarks are used: First, we compare the runtime performance of our system using SPECint2006 [29] with the reference workload. Second, we evaluate the performance of popular server applications following the same approach used by Lvin et al. [18].

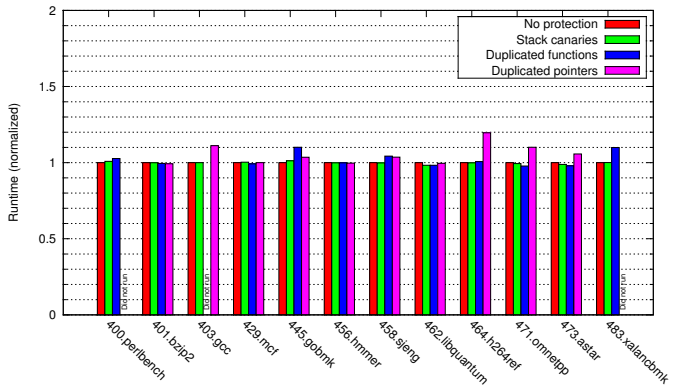


Fig. 5. Runtime performance of the SPECint2006 benchmark suite

SPECint2006. Figure 5 presents the normalized runtimes of the 12 programs of SPECint2006. For some of the programs, we were unable to produce valid runs: (a) 403.gcc did not compile when using duplicated functions because of out-of-memory issues (b) 400.perlbench and 483.xalancbmk did not run using

duplicated function pointers because of false positives. These issues will be further discussed in Section 5.

As observable in Fig. 5, function duplication incurs no overhead for most programs and even causes a slight speed-up in some applications. There are some programs for which there is overhead when using duplicated functions but this overhead is less than 10% in all cases. The use of duplicated function pointers can cause a higher overhead for applications that make heavy use of function pointers. The most affected program, 464.h264ref, has an overhead of approximately 20%. While this is a non-negligible overhead, it is also obvious that all other applications that make more moderate use of function pointers have a much lower overhead.

Server Applications. The performance of three different server applications was measured: the thttpd web server, the bftpd FTP server and the OpenSSH server. For the first two, we measured the time it takes for 50 simultaneous clients to make 100 requests each. For OpenSSH, we measured the time needed to authenticate, spawn a shell and disconnect.

Figure 6 shows the results of these experiments, normalized with the programs' unprotected versions. The overhead incurred by duplicating functions is negligible. The overhead due to function pointer duplication is less than 2% in almost all cases with the exception of the OpenSSH server where the overhead is 6%.

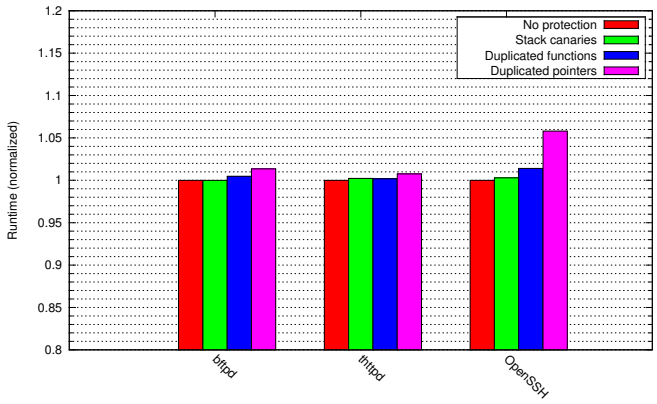


Fig. 6. Runtime performance of some server applications

4.3 Code-size evaluation

The duplication of all functions so as to eliminate the need of return addresses leads of course to the expansion of the code section of a program. The average size of the SPECint2006 binaries, before function duplication is 2MB. After the

application of our duplication algorithm, the average size is 156 MB. While this may appear as a very high overhead, it is important to remember that this is the size of the program on disk. Given the capacity of today's commercial off-the-shelf hard disks, we believe that the difference in size is unimportant. The enlarged code section will still be loaded as individual memory pages by the operating system, and each page will be in-turn swapped-out when it is no longer necessary. This behavior allows for the majority of the code to be off-memory, on-disk while still maintaining low performance overheads, as shown in Section 4.2.

5 Discussion and Future Work

Return Address Protection. In principle, the technique of function duplication offers complete protection from return address smashing since the return addresses are simply removed from the program. However, special care has to be taken to ensure that the entire attack surface is covered. As explained in Section 2.2, return addresses cannot be removed from functions that have external linkage. This means that there will always be one copy of such functions that will not benefit from our protection. Fortunately, this problem can easily be solved by postponing the function duplication step to link time where the linkage type of all function can be changed to internal.

Functions that are called though function pointers pose a similar problem as those with external linkage. For these functions it is also impossible to know at compile time where they will return to, so we need to keep one copy of each which still uses a return address. Unfortunately, there seems to be no easy way to solve this problem, leaving those functions unprotected in our current implementation. It should be noted, however, that these function *are* protected when a direct call to them is made. This means that if such a function contains a vulnerability, it can only be exploited when it is called through a pointer.

Since our solution to the problem of recursive functions, discussed in Section 2.2, introduces state on the stack, one might wonder whether this state could become a new attack vector. As mentioned earlier, our approach is similar to that used in [15] with the difference that the indices used by us are local to functions instead of global to the program. Because the indices are local to functions, the number of possible legal values an attacker could overwrite an index with, is very limited making it practically impossible to construct exploits of any real value. If the index on the stack is not in the range of values expected by the program, the process aborts thus any in-place attacks are terminated.

Lastly, note that our system focuses on the protection of return addresses instead of the detection of buffer overflows. Since the return addresses cannot be modified by an attacker, applications may be able to execute correctly even in the presence of an overflow that would normally hijack or crash the program.

Memory Usage While duplicating functions, memory usage can be problematic: some programs may fail to compile due to out of memory errors. However, since most software is provided in binary form, a vendor can provide the necessary

resources once upon compilation and then ship the compiled and protected binary to all users. Nevertheless, to make our countermeasure as widely applicable as possible, we currently provide two possible solution to this problem. The first is the ability to manually exclude functions from the duplication process. While not optimal, this approach can be used by seasoned developers who understand which functions are the least likely to contain vulnerabilities. One of our goals however, is to make the protection of return addresses fully automatic. Hence, we are currently implementing an automatic safety analysis framework to eliminate the need of manual intervention.

In our current implementation, a function is considered unsafe if it might do anything that alters memory in an unexpected way. This approach is already more sophisticated than the heuristics used in most stack canary implementations, which only check for the presence of character buffers on a function's stack frame. Our framework takes into account the fact that the return address of a function other than the current one may be overwritten by writing through a compromised pointer [6]. This means that any store through a pointer may only be considered safe if it can be proven where that pointer points to and that the corresponding memory location is large enough to store the value being stored.

The results of this framework are looking promising in the sense that a lot of functions can already correctly be detected as being safe. However, a significant amount of work still lies ahead since the number of functions that can be eliminated from duplication is still not enough to make all complex programs compile in a safe way.

Memory Aliasing Most low-level programming languages allow casting of pointer types. This makes it impossible to know in some cases whether or not a store to a memory location stores a function pointer. In our current implementation, we simply ignore loads of pointers to which no corresponding store has occurred which means that such pointers are not protected.

One way to protect those pointers is by observing that, if a function pointer is aliased by another pointer, this pointer is a `void` pointer in most cases. This means that we could duplicate all `void` pointers in addition to function pointers although this would likely incur a higher runtime overhead than the current function pointer duplication.

6 Related Work

In general, there are three categories of countermeasures against popular control-flow hijacking attacks: (a) specifically trying to protect return addresses and function pointers, (b) more generally trying to counter vulnerabilities that enable the overwriting of sensitive control-data structures, and (c) minimizing the usefulness of being able to overwrite a control-data structure. Our function-duplication technique belongs to category (a). Here we will discuss some of the most well-known existing countermeasures in all three categories.

A popular way of detecting attacks against a function's return address is through the use of stack canaries. Most modern compilers have implemented some form of stack canaries (e.g., StackGuard [9] for GCC). ProPolice [11] is a re-implementation of StackGuard with the addition of security-enhancing features such as variable reordering. While more robust than StackGuard, ProPolice also cannot detect a return-address overwrite, happening through an indirect pointer overwrite.

Another way of protecting return addresses is by using a shadow stack. Return addresses will be pushed on both the normal and the shadow stack and then checked for equality when used. This approach is used by the Return Address Defender [7] and Libverify [4] and in both cases, the mechanism necessary to protect the shadow stack incurs a significant performance overhead. StackShield [35] follows a similar approach with a return-address shadow stack, but also attempts to protect function pointers by verifying that they do not point within the program's stack, heap or data segment. Recent attacking techniques however, like return-to-libc [10] and return-oriented-programming [26], that do not need to inject new code in a process' address space circumvent this countermeasure.

Function pointers can be protected much in the same way as return addresses. PointGuard as discussed in Section 1, is a technique specifically designed to protect function pointers. ValueGuard, by Van Acker et. al [34], protects all data items from overflows, including function pointers. This is done by placing canaries before every item, similar to the StackGuard approach. Unfortunately, the large number of checks causes this technique to incur a large runtime overhead.

The most common vulnerability that leads to the attacker being able to overwrite the return address or a function pointer located on the stack is a stack-based buffer overflow. A popular way of defending against buffer overflows is through the use of bounds-checkers, systems that discover the correct bounds of each object and terminate programs that write out-of bounds. These systems provide strong security guarantees and researchers have therefore proposed a plethora of bounds-checkers [3, 12, 13, 14, 19, 20, 30]. These security guarantees however, usually come at the cost of high runtime overheads. Today, the fastest implementation of a bounds checker has a runtime overhead of about 60% [2].

The third type of protection tries to prevent the usefulness of overwriting a return address. An attacker can make the most out of overwriting a return address if he can point it to code he supplied himself. The easiest way to do this is to inject code together with the new value for the return address which means this code will end up on the stack. An obvious way to prevent such an exploit is to make the stack non-executable [33]. Another common way to mitigate such exploit is Address Space Layout Randomization (ASLR) [5, 32]. Both approaches however, have their limitations: a non-executable stack will not prevent overwriting the return address with the location of existing code in the program's address space and ASLR can be de-randomized [23, 27] or bypassed altogether [25].

When a control-flow hijacking attack is being carried out, the attacked program inevitably diverges from its normal control-flow. A number of countermeasures

have been proposed that use this fact to detect control-flow hijacking attacks. One way to detect such divergences is to monitor the system calls made by the program. Systrace [22], by Provos, automatically builds system call policies during training sessions. If, during a normal run, a system call is encountered that is not specifically allowed by the policy, the user is asked if the call should be allowed. Linn et al. proposed statically analyzing binaries to discover from which addresses system calls are made [16]. These locations are then transfered to the kernel which enforces them on subsequent system calls.

The previous techniques only try to enforce normal program behavior at its boundaries, i.e., the behavior that is exhibited towards the kernel. Abadi et al. have proposed a more complete technique for insuring the integrity of control-flow within programs [1]. They statically create the program's Control-Flow Graph (CFG), which contains all control transfers the program can legally make. Then, the binary is instrumented so that every control transfer is checked at runtime to correspond to an edge in the CFG. A similar approach is taken by Philippaerts et al. in [21]. Here, control-data is masked before being used in order restrict the addresses it can hold.

7 Conclusion

The market penetration of personal computing devices, and the expansion of server-side computing resources to handle the fast-growing client population, has made the task of protecting vulnerable software and private user-data more relevant than ever. In this paper we reviewed the shortcomings of popular countermeasures against control-flow hijacking and introduced the concept of program-part duplication as a means of securing the control-flow of a potentially vulnerable program.

We instantiated the duplication concept and introduced a novel technique that duplicates the functions in a program in such a way that return addresses are no longer needed, providing a complete protection against return address smashing attacks. Additionally, we described a generic method which uses duplication to protect all types of control-data and demonstrated how this method can be used to protect function pointers. The evaluation of our techniques showed that they provide a very effective protection while incurring only a minimal runtime overhead in real-life applications, making them applicable to both desktop and server environments.

Acknowledgments. This research was done with the financial support from the Prevention against Crime Programme of the European Union, the IBBT, the IWT, the Research Fund KU Leuven, and the EU-funded FP7 project NESSoS.

References

1. Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, November 2009.

2. Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th conference on USENIX security symposium*, SSYM'09, pages 51–66, Berkeley, CA, USA, 2009. USENIX Association.
3. Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *ACM Conference on Programming Language Design and Implementation*, 1994.
4. Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Technical Conference*, San Diego, California, USA, June 2000.
5. Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington, D.C., August 2003.
6. Bulba and Kil3r. Bypassing Stackguard and Stackshield. *Phrack*, 56, 2000.
7. Tzi-cker Chiueh and Fu-Hau Hsu. RAD: A Compile-Time Solution to Buffer Overflow Attacks. In *ICDCS'01*, pages 409–417, 2001.
8. Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard™: Protecting Pointers from Buffer Overflow Vulnerabilities. In *In Proc. of the 12th Usenix Security Symposium*, 2003.
9. Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, 1998.
10. Solar Designer. Getting around non-executable stack (and fix). Posting to BuqTraq mailing list http://seclists.org/bugtraq/1997/Aug/63, August 1997.
11. IBM. Gcc extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/.
12. Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *3rd International Workshop on Automatic Debugging*, 1997.
13. Samuel C. Kendall. Bcc: Runtime Checking for C Programs. In *USENIX Summer Conference*, 1983.
14. Kyung-Suk Lhee and Steve J. Chapin. Type-Assisted Dynamic Buffer Overflow Detection. In *11th USENIX Security Symposium*, 2002.
15. Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with "Return-Less" kernels. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 195–208, New York, NY, USA, 2010. ACM.
16. Cullen Lin, Mohan Rajagopalan, Scott Baker, Christian Collberg, Saumya Debray, and John Hartman. Protecting against unexpected system calls. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, Maryland, August 2005. USENIX Association.
17. LLVM Developer Group. The LLVM Compiler Infrastructure. http://llvm.org/.
18. Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Archipelago: trading address space for reliability and security. *SIGOPS Oper. Syst. Rev.*, 42:115–124, March 2008.
19. Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure: Progress Report. In *International Symposium on Software Security 2002*, 2002.
20. Harish Patil and Charles N. Fischer. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Software: Practice and Experience*, 27(1), 1997.

21. Pieter Philippaerts, Yves Younan, Stijn Muylle, Frank Piessens, Sven Lachmund, and Thomas Walter. Code pointer masking: hardening applications against code injection attacks. In *Proceedings of the 8th international conference on Detection of intrusions and malware, and vulnerability assessment*, DIMVA'11, pages 194–213, Berlin, Heidelberg, 2011. Springer-Verlag.
22. Niels Provos. Improving host security with system call policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, Washington, D.C., August 2003.
23. Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *25th Annual Computer Security Applications Conference*, 2009.
24. Dan Rosenberg. Breaking LibTIFF. http://vulnfactory.org/blog/2010/06/29/breaking-libtiff/.
25. Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. Q: Exploit hardening made easy. In *Proceedings of the USENIX Security Symposium*, 2011.
26. Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Sabrina De Capitani di Vimercati and Paul Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.
27. Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307, Washington, D.C., October 2004.
28. Eugene H. Spafford. The internet worm program: an analysis. *SIGCOMM Comput. Commun. Rev.*, 19:17–57, January 1989.
29. Standard Performance Evaluation Corporation. SPEC CINT2006. http://www.spec.org/cpu2006/CINT2006/.
30. Joseph L. Steffen. Adding Run-Time Checking to the Portable C Compiler. *Software: Practice and Experience*, 22(4), 1992.
31. Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, EUROSEC '09, pages 1–8, New York, NY, USA, 2009. ACM.
32. The PaX Team. Documentation of ASLR in PaX. http://pax.grsecurity.net/docs/aslr.txt.
33. The PaX Team. Documentation of PAGEEXEC in PaX. http://pax.grsecurity.net/docs/pageexec.txt.
34. Steven Van Acker, Nick Nikiforakis, Pieter Philippaerts, Yves Younan, and Frank Piessens. ValueGuard: Protection of Native Applications against Data-Only Buffer Overflows. In Somesh Jha and Anish Mathuria, editors, *Information Systems Security*, volume 6503 of *Lecture Notes in Computer Science*, pages 156–170. Springer Berlin / Heidelberg, 2011.
35. Vendicator. Stack Shield technical info file v0.7. http://www.angelfire.com/sk/stackshield/, January 2001.
36. John Wilander, Nick Nikiforakis, Yves Younan, Mariam Kamkar, and Wouter Joosen. RIPE: Runtime Intrusion Prevention Evaluator. In *In Proceedings of the 27th Annual Computer Security Applications Conference, ACSAC*. ACM, 2011.