Seminar: Control Flow Integrity based Security

Report on: Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM

Patrick Funke

Institute of Informatics
Technische Universität München, Germany
July 10, 2016

Abstract. Since the widespread adoption of stack protection mechanisms, attacks on software's control flow focused more and more on the forward edges. To cope with this issue, the authors present three tools, two of which harden the output binary and the third one is a tool for developers to prevent vulnerabilities. They implemented a C++ vtable verification for GCC, an indirect function call checker for LLVM and an addition to LLVM's UBSan. Their new mechanisms improved on the performance overhead that prior work and put great effort into creating usable tools for the real world. Providing a realistic attack model they were able to bring a certain level of security to every day software. More recent work developed their ideas further which makes these tools a starting point for current approaches and improvements on control flow integrity.

1 Introduction

Securing the control-flow integrity (CFI) is a rather new approach to further improving the safety of software, especially of programs like webbrowsers which are complex, widespread and often running with high privileges. No software is free of errors and certain bugs (e. g. use-after-free, buffer-overflows) can be abused in such a way that the program's control-flow is altered. Attackers use these possibilities as an entry point for further taking over an application. Thus, different approaches to hardening that kind of attacks emerged over time.

There are three types of instructions that alter the control-flow and therefore are in the focus of research. The first are the return addresses, stored on the stack. The other two are virtual and indirect function calls, which both operate on the heap. They are also referred to as forward-edges in the control flow, as they are a forward jump into another function while the return after executing a piece of code is analogously called a backward-edge.

Protection of stack data and return addresses is a widespread method for increasing computer security - in scientific contexts as well as in production environments, because most mainstream compilers offer such tools. As a result, recent attacks exploit heap-based memory corruption bugs and compromise the control-flow integrity (CFI). Protecting the program-control data on the heap however appears to be a complex problem and is a current research subject. The authors recognize that this kind of CFI enforcement has thus not been adopted in mainstream compilers. Most approaches are either ad-hoc mechanisms, binary rewriting frameworks or even only experimental, also incremental compilation and dynamic loading are often impossible. As a result, using this kind of tools in production compilers is not practicable.

The authors present two mechanisms they implemented to protect the CFI while not restricting compiler optimizations, operation modes, or features like Position-Independent Code (PIC) and C++ exceptions. Also they do not re-

strict the execuction environment of the final binary such as dynamically-loaded libraries or Address Space Layout Randomization (ASLR). The result are fully functional tools for the respective production compilers that are shown to be practical and at the time of the publication very efficient. Also, they describe and resolve the challenges that arise when developing a CFI solution usable in a real-world environment.

In their paper, the authors focus on verifying the targets of forward-edges of indirect control transfers at different levels of precision, which depends on target's type and applied analysis. In C++, virtual calls make up the largest share of indirect control transfers, so the first approach protects these virtual calls by verifying the vtables. Evaluation of the binary compiled with the CFI tools showed that 95% to 99.8% of all indirect function calls could be protected, depending on the tool. The performance penalty ranges from 1% to 8.7%, evaluated with the SPEC CPU2006 benchmark suite and web browser benchmarks, all performed on the recompiled version of the Chromium browser.

At the time, the presented tools compared well to other research prototypes in terms of performance. Although the provided level of security is not as good as from these other prototypes, they still provide strong guarantees for the defined realistic attack model. Concurrent other tools like MIP [7] and CCFIR [12] have a significantly higher overhead or must sacrifice practicability to achieve a better performance, like SafeDispatch [6].

2 Attacks and Compiler-based Defenses

Altering the control flow of software is a powerful opportunity for attackers to achieve their goals. Escpecially complex software running with high privileges like web browsers is a common target, as the attacker can alter the control flow while having a lot of privileges on the machine. Typically, these attacks exploit vulnerabilities at a low level which brings importance to techniques such as memory protection, ASLR and fine-grained type-safety guarantees of high-level languages. Modern compilers, operating systems and runtime libraries offer these and for instance, set executable code in memory as not writable or verify stack integrity by checking whether secret values, commonly referred to as stack canaries, have been modified. All of these techniques are widespread and thus, stack-based attacks have become more rare. Heap-based attacks on the other hand have increased, as the heap remained hard to protect. Control flow data on the heap, expecially C++ vtable pointers or function pointers are a common target. Corresponding attacks are known as return-oriented programming or "return-to-libc" as an attacker might try to return to a place in the code that was not intended and can be reused for malicious activity.

As a solution, the authors created two tools for mainstream compilers that help protect the control flow integrity by hardening the resulting binaries against

heap-based attacks. They focus especially on the forward-edges of the control flow, namely function pointers and vtable pointers. This is done by restricting these pointer targets to a set of valid targets which are determined by the compiler. A third tool is presented for improving the development of software, by analysing it and trying to find possible control flow violations before publishing an application. The mechanisms have been implemented for the GCC and LLVM compilers and have - apart from some differences in security and implementation details - in common that they:

- add new, read-only metadata to compilation modules to represent aspects of the programs static CFG;
- add machine code for fast integrity checks before indirect forward-edge, control-flow instructions;
- optionally divert execution to code that performs slower integrity checks in certain complex cases;
- call out to error handlers in the case of failures; and
- may employ runtime library routines to handle important exceptional events such as dynamic loading.

Also, they follow a very specific attack model for their defenses. The first assumption, like in comparable work, is that all writable program data may be corrupted at any time. On the contrary, read-only data, the stack and thread register state are assumed to be safe and may not be modified. For this model to be applicable, other mechanisms - especially stack defenses must be in place. Otherwise, the presented software can be bypassed. While other work tries not to rely on the trustworthiness of the used compiler, the authors chose to make use of the compiler toolchains abilities to reduce complexity, increase portability and particularly allow optimizations. Although they note the importance of questioning the trust in a compiler, they do not see concrete benefits of distrust in the compiler when developing a compiler-based CFI mechanism.

The result were efficient and most notably usable tools for production environments. At the time the report was written, there was a number of approaches to CFI available, but they did not provide the usability one would expect from a production compiler. Other work struggled with incremental compilation and dynamic libraries as well as very high performance overheads. To cope with these issues, more coarse-grained techniques had been presented [7] [12], but they came at cost of less security. By properly integrating their mechanisms into compilers and putting their focus on forward-edges only, the performance overhead improved a lot on prior, different approaches while providing a defined level of security.

2.1 Related Work

Up until the paper was published, many very different approaches for enforcing CFI have been described and implemented, including XFI [4], BGI [2], HyperSafe [9], CFI+Sandboxing [11], MoCFI [3], CCFIR [12], Strato [10], bin-CFI [13],

MIP [7] and SafeDispatch [6]. Many only enforce coarse-grained CFI which has better performance but is less secure. Furthermore, a part of these defenses is not compiler-based but uses binary rewriting. Other compiler-based defenses are most importantly HyperSafe, MIP, CFI+Sandboxing and SafeDispatch, with the latter being the most similar concurrent approach. It adds runtime CFI checks and reduces runtime overhead by profile-driven, whole-program optimization. At every call site, vtable pointers are verified and the address in the vtable is checked to be a valid method for the corresponding call site. The major drawback of SafeDispatch however is the impossibility of separate compilation and dynamic libraries which makes it very impractical.

VTV, one of the tools the authors developed and implemented, was the starting point for additional work and its protection scheme was widely adopted. VTV has been recently refined by Haller et al. [5] to further improve its strictness by changing the decision policies and therefore increasing security. The idea behind VTV has also been used in VTable Interleaving [1] which implements similar validity checking for vtable pointers by laying out the vtables in a manner that a simple range check of the pointer suffices to determine whether it is valid or not. As a result, the performance overhead for this similar level of security is very low compared to the original VTV.

3 VTV: Virtual-Table Verification

The first mechanism the authors present is VTV which is short for vtable verification. It's a CFI tool implemented for the GCC C++ compiler. It only verifies virtual calls, but not other types of indirect control flow. The authors claim virtual calls to be the most common kind of indirect calls which makes their work applicable for this most attractive target.

3.1 Problem Description

Virtual function tables are the way of implementing polymorphism in C++. If a base class and a child class declare the same function, they are declared as virtual functions. Every class has a corresponding vtable that stores pointers to the actual implementation of every function an instance of that class provides. At runtime, when such a virtual function is called, the program uses this vtable to get the address of the correct implementation and jumps to it. Contrary to the actual vtables, which are placed in read-only memory, the objects referring to these vtables are found on the heap. By exploiting use-after-free bugs, the vtable pointer can be altered and lead to an invalid vtable, either reusing an exisiting one that contains functions which are not intended at this point or creating a new vtable, although this may be hardened by additional mechanisms like DEP (Data Execution Prevention) in place. When such object functions are then used, the attacker's code is executed.

3.2 Overview and details of VTV

VTV inserts checks at every call site of a virtual function. Before executing the call, the pointer is verified to be in the set of either the VTable for an object's static type or one of its descendant classes. If that's not case, the call is not executed at all. This is done by changing the intermediate representation code, which is an interstage between the original source code and the machine instructions, for every virtual call. VTV inserts a verification call after retrieving the vtable pointer value out of the object, ensuring that the value that's being checked is not changed between the retrieval and the verification call. The verifier function checks the objects vtable pointer against the set of valid vtables provided by the compiler. It returns the correct pointer from the vtable if everything is fine and calls a failure function that reports an error and aborts the execution immediately otherwise.

An important improvement to earlier work is the possibility of incremental compilation and dynamic loading, which is often denied in other approaches. Prototypes and primarily academical work often does not take care of these dealbreakers for real world applications. When allowing them, the entire class hierarchy of the program is not available for the compiler, as parts of it may reside in precompiled libraries and modules. As a solution, VTV consists of a compiler piece and a runtime library called libvtv, which are both part of GCC. In addition to the insertion of verification calls, VTV collects information about the class hierarchy and the vtables to supply generator functions that build the complete sets of valid vtable pointers at runtime, which is the key to allowing dynamic loading. A pointer to the set of valid vtable pointers is stored in a vtable-map variable for every polymorphic class which is then passed on to the verification function. The vtable-map variables must be set to the pointer of the corresponding vtable set before any virtual call is performed, so they are initialized by special init functions that run before other standard initialization functions which ensures that the necessary data is in place when the first virtual call is made.

Although the vtable-map variables and generated vtable-pointer sets should be read-only during runtime, they must be writable for some time as they are built by generator functions when the program starts or when a library is dynamically loaded. Although the authors do not describe such an attack, the mechanism is obvioulsy vulnerable during this time, so it should be as short as possible. Based on mmap, the authors created their own memory allocation scheme for the vtable-pointer sets which makes finding all these sets in memory easier for VTV. Vtable-map variables are stored in a special section in the executable that is page-aligned and padded with zeros up to the end of the memory page to prevent other data from being on that same page. For an update of the vtable-map variables and vtable pointer sets VTV makes all corresponding pages writable, while using pthread mutexes to avoid interference of other threads. After the update, all pages are set back to read-only.

3.3 Practical Experience with VTV

When trying to achieve global unique vtable-map variables for every class, the authors realized that in some use cases the compiler is not able to ensure this global visibility; resulting in multiple instances of a vtable-map variable. These different instances might represent different and incomplete sets of valid vtables which leads to incorrect aborts. To work around this issue, the authors added a mechanism to keep track wheter VTV has already seen another instance of the vtable-map variable for a given class. Only if not, VTV will create a new vtable-map.

Another issue is the mixing of verified and non-verified code. Especially when using external libraries or plugins that are not verified with VTV, verification errors occurred. To deal with these, a replaceable failure function is called when the usual verification function failed. Programmers may write these replacement functions as a kind of whitelist for non-verified segments. When a call check fails in the first place, this replacement function goes through an array of allowed address ranges. If the call target is found there, execution continues. Because libraries can be dynamically loaded, the whitelist arrays have to be built when such a library is loaded and before it is called for the first time. The authors do not describe how much additional work for the programmer is necessary when using these replacement functions for a large application.

3.4 Alternatives & Enhancements for VTV

As the VTV's performance was rather good in their first implementation, the authors did not put effort into improving it. But they recognize various points of possible optimization such as partial inlining of the verification call sequences, because the call overhead makes up a big part of the performance penalty. Secure methods for caching and reusing frequently verified values could be another starting point for optimizations. They also describe the possibility of altering other toolchain elements, apart from the compiler, to sort out ordering issues between the generator functions for vtable-pointer sets and functions performing virtual calls.

4 IFCC: Indirect Function-Call Checks

Similar to VTV, Indirect Function-Call Checks (IFCC) is a CFI tool the authors implemented for the LLVM compiler. It operates on the intermediate representation during link-time optimization and does not only protect vtables, which are specific to C++, but all indirect calls. By generating jump tables for indirect-call targets and transforming the function pointers in a manner that they now point to a jump table IFCC ensures, that a program may only jump to valid targets. A function pointer not pointing into the appropriate table is considered a CFI violation and IFCC will try to force the function pointer into the correct

jump table. Like VTV's vtable-pointer sets, IFCC builds sets of valid function pointers, represented by jump tables.

Because every indirect call is checked against a jump table, IFCC reduces the set of possible jump targets significantly and thus limits an attackers options. A jump table entry is merely an aligned jump instruction to a function, placed in read-only memory. By padding the table to a power-of-two size any aligned jump to the padding will cause a crash. The power-of-two size of the tables allows IFCC to use a mask to force function pointers into the right set on every call site.

Similar to VTV, IFCC secures indirect function calls by building so called jump tables, where valid targets for these indirect calls are stored. On the call site, the pointer is replaced by the corresponding pointer to the jump table where the valid function pointers are stored. By padding the jump tables with trap instructions and transforming the pointers in a corresponding way, incorrect jumps will cause the program to crash. Pointers to functions that have their address taken, are transformed to pointers to a jump table entry. Furthermore, indirect calls are replaced with a check against the function-pointer set corresponding to the call site. The checks are performed using a mask and a base address, in a way that a manipulated pointer will target a trap instruction or at least a target that is inside the valid set. IFCC offers different levels of precision for the function-pointer sets, because the most precise imaginable (one function-pointer set per function type signature) might fail in real world applications as they do not always respect function pointer types. The authors present two methods for assembling function pointer sets, one (called Single) putting all the functions in a single set and the other one (called Arity) assigning functions to a set according to the number of arguments passed to the indirect call. In theory, the method to construct the function pointer set can be chosen individually for each function, as long as there is exactly one table per call site. This restriction should be met regardless of the analysis technique.

4.1 Practical Experience with IFCC

External code poses a problem for IFCC, too, because a function that was not declared or defined during link-time optimization will not be found in the jump table and therefore recognized as a CFI violation. This happens for example with JIT code (e.g. JavaScript engines) that can generate functions dynamically, external functions returning external function pointers and of course function that is passed to an external library and then called there with external function pointers. Depending on the kind of code, the amount of these false positives varies greatly from extremely frequent with JIT-generated code to very rare when working with signal handlers for instance.

To cope with this issue, the authors introduced a fixed-size, writable array in the beginning of each jump table. IFCC then rewrites calls to external functions that return function pointers and saves the pointer into the writable array if it

is not already stored there; the replacement function then returns a pointer to this array entry. When the array is full, the program halts with an error. With this technique, function pointers from external code are turned into jump table entries, the only drawback being the addition of a few writable function pointers. By further optimizations, this memory could also be protected. Although the authors describe these optimizations, they did not implement them for their prototype because of time constraints.

4.2 Annotations

For further improving usability in production environments, the authors implemented a second version of IFCC that is intended to become a part of LLVM. Instead of forcing the program's contol flow into a crash or towards any pointer in the valid set if the check fails, they pass this pointer to a custom failure function which handles the error, similar to VTV. Additionally, annotations and a simple dataflow analysis have been added to the implementation in order to improve detection and handling of problems.

5 FSan: Indirect-Call-Check Analysis

The third tool the authors developed does not directly harden software, it is a tool intended for development. The idea behind such a tool is the trade-off between security and correctness of the output. If a tool is uncertain about whether the control flow is correct or not, it has to give in and skip this instruction; if it inserts a check nonetheless, the result is broken because a correct instruction has been disabled. Thus it appears that code with less uncertainties for a CFI tool increases its security.

FSan is an optional indirect call checker and has been added to Clang's undefined behavior sanitizer (UBSan) in version 3.4 which is designed to catch a variation of undefined behaviors - not only in a control-flow context [8]. The tool is designed for improving applications by checking at runtime whether or not the function types of caller and callee match and thus detects CFI violation for indirect function calls. As defined in the C++ standard, the outcome of calling a function through a pointer to a function type is not defined, given that the types do not match which may lead to a security issue. FSan works with executable metadata prepended to every function. On an indirect call site, the first four bytes from the function pointer are loaded and compared to the expected signature, followed by checking the Run-Time Type Information (RTTI) pointer from the next 4 or 8 bytes if the first check was succesful. In case one of these checks fails, an error message is printed.

The paper notes that the check in its present form only works for C++ and is not as accurate as VTV, because it only validates the function's type - not the programs control flow graph.

5.1 Practical Experience with FSan

When recompiling the Chromium web browser with FSan enabled, a variety of undefined-behavior reports emerged. With the two main kinds of these problems have a simple fix, some problems could be fixed by refactoring a part of the Skia graphics library. The remaining problems require more dedicated work to fix; they are more widespread and a fix is not as obvious.

6 Security Analysis

While other papers seemingly try to beautify the already difficult to quantify security analysis with picky choices in terms of the software they apply their tools to, the authors recognized that it makes sense to test CFI mechanisms with complex, real world software that may be a valuable target. The chromium web browser is such an application. By recompiling it with VTV and GCC, IFCC and Clang (LLVM) it was possible to compare the performance and security to the unhardened version. Notably, they evaluated the whole final output binary and not only the output of the single pass their tool operates in which takes later optimizations and the linking into account.

By calculating the Average Indirect-target Reduction (AIR) introduced by Zhang and Sekar [13] they quantified their results. The AIR metric is calculated as follows:

$$AIR = \frac{1}{n}\sum_{i=1}^{n}(1 - \frac{T_i}{S}) \tag{1}$$

n: number of indirect control-transfer instructions (indirect calls, jumps, and returns), Ti: number of instructions the ith indirect control transfer instruction could target after applying a CFI technique, S: size of the binary

They limited the number of indirect control-transfer instructions to forward-edges in the control flow only. This excludes returns since they are assumed to be protected by stack-based defenses and not subject of the research. This modified AIR metric is called fAIR. Although it is a notable effort to apply some metric - AIR still remains to be one of the more commonly used - it's relevance is questionable. Even if the number of allowed indirect targets is very low, which leads to a high AIR value, this limited set of possibilities may still contain the one that is needed in order to perform an attack. How valuable or usable an instruction is for the possible attacker is not taken into account at all, so when interpreting the resulting numbers this limitation has to be kept in mind. Even a CFI technique with a very high AIR value can produce a final binary as vulnerable as before, because the dangerous call target has not been detected.

VTV protects 95.5% of Chromium's indirect control transfer instructions. Of these targets, about 10% may be spilled to the stack. Since the presented method does not take care of the already widely protected stack, this does not bring up a security flaw in the VTV system. When using VTV in a production environment, these additional protections must definitely be applied. IFCC performs even

better on Chromium's codebase and is able to protect 99.8% of the instructions in question. Here, about 18% of the call targets may be spilled to the stack. In both cases, most of the unprotected targets stem from C libraries or C-style callbacks. The ROP-gadget count is another common measure, but as VTV and IFCC do not take care of backward edges (returns) at all, it is obsolete.

7 Performance Measurements and Results

VTV's and IFCC's performance were measured by recompiling the Chromium web browser with these additions and comparing its performance in benchmarks to the insecure version. The authors used the SPEC CPU2006 benchmark suite and the Dromaeo, SunSpider and Octane browser benchmarks. CPU turbo mode and ASLR have been turned off on the testing machine because they produced too much variation in the results. The Chromium web browser was chosen for its widespread real world usage and complexity, which makes the results more applicable to other software. In particular, it contains hundreds of thousands of virtual calls, links many non-verifiable third-party libraries and extensively uses dynamic library loading. All together Chromium represents real life attack targets very well.

7.1 VTV Performance

A subset of these benchmarks which execute a lot of indirect calls and thereby show significant results were chosen and performed. Initially, VTV's performance penalty showed to be 19.2% in the worst case. The authors then looked further into the reasons behind the performance penalty. With the tool inserting additional instructions, some penalty is not avoidable. By replacing the libvtv functions with stubs, they measured the minimum overhead created by the insertion of the additional check instructions at every call site. For the benchmarks in question, the minimum penalty ranged from 1.0% to 4.7%. By using profile guided optimizations of the GCC compiler which reduce the amount of virtual calls and statically linking libvtv, the penalty of the complete version of VTV could be reduced to 10.8%. Other benchmarks executing less virtual calls showed even fewer performance penalties.

7.2 IFCC and FSan Performance

Like VTV, IFCC's performance differs greatly according on the number of indirect calls and the version of IFCC itself. Depending on whether the jump table fits into one page in memory, either 14 bytes or 20 bytes of instructions are added to every call site. FSan's impact changes with the used optimization level of the compiler, the authors found that it adds roughly 12 instructions per call site. They compared the benchmark results for the same benchmarks used to test VTV's performance to a version built with the bare Clang LTO. Both IFCC and FSan showed a performance loss of about 4% in both Dromaeo and SPEC

CPU2006. IFCC's operation mode (Single or Arity) made no significant difference. To conclude, the number of indirect calls is directly proportional to the performance overhead.

7.3 Comparison to Prior Work

Regarding the SPEC Perl benchmark, IFCC outperforms prior work by far. In terms of CFI security, Perl is very difficult to come by because it interprets every instruction by making an indirect call. CFI implementations at the time such as CCFIR by Zhang et al. [12], bin-CFI by Zhang and Sekar [13], Strato by Zeng et al. [10], and MIP by Niu and Tan [7] hat performance penalties ranging from 8.6% to 31.3% for this particular benchmark. IFCC however only produces less than 2% overhead. As a reason, the authors refer to their focus on forward edges only and little changes to the possibilities for the compiler to apply optimizations. Concluding, the achieved level of security may be different, but more in line with the expectations of such protections in real applications.

8 Conclusions and Discussion

This approach's major improvement for CFI consists of the focus on producing tools that can be used in real world environments. All three are compiler-based mechanisms and have been implemented for a mainstream compiler. VTV secures virtual calls in C++ by ensuring that a vtable is a vtable for the program and also semantically correct for a specific call site; it has been added to GCC. IFCC has been implemented for the LLVM compiler and checks whether the target of an indirect call is one of the address-taken functions in the program. FSan does not harden the final output binary but is a developer's tool to check for indirect calls whose function signature does not match its runtime target. Each of the tools was able to produce working versions of the chromium browser and provide a certain level of security. Although the AIR metric is questionable as mentioned in 6, it shows that both VTV and IFCC drastically limit an attacker's possibilities while keeping the performance overhead within the scope of roughly 10% which was at the time better than other approaches. Most notably, they allow dynamic loading and incremental compilation which is necessary for productive work. Most other CFI tools were merely prototypes or introduced these impractical restrictions.

The attack model is accurately described but passes some responsibility on to other techniques. Most important, the authors focus on forward-edges in the control flow only - so return adresses are not protected. VTV and IFCC have a few remaining security issues that can be dealt with by further optimization and using a strong and reliable stack protection. All in all they provide at least a good reason or a proper solution for their vulnerabilities. Additionally, perfect security was not the point of the paper. Instead, the result should be a working tool suitable for the real world that makes software more secure than before.

When testing the performance and security of their mechanisms, the authors limit their testing to a single application - contrary to similar work. An interesting addition to the results would be measuring the performance of more than one applications and comparing the results. Different software may behave differently and the impact on performance could be a lot higher than for the Chromium browser. The performance is claimed to be better than prior work but - especially compared to recent efforts - it is not as impressive either. The worst case performance penalty showed to be almost 20% and the most effective part of improving VTV's performance was reducing the overall number of virtual calls by compiler optimizations. This approach does not improve the actual performance of VTV and thus seems more like beautified numbers. Still, other CFI mechanisms at the time performed a lot worse which makes VTV still an important improvement.

References

1. Bounov, D., Kc, R.G., Lerner, S.: Protecting c++ dynamic dispatch through vtable interleaving. In: Symposium on Network and Distributed System Security (NDSS) (2016)
2. Castro, M., Costa, M., Martin, J.P., Peinado, M., Akritidis, P., Donnelly, A., Barham, P., Black, R.: Fast byte-granularity software fault isolation. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles. pp. 45–58. ACM (2009)
3. Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nürnberger, S., Sadeghi, A.R.: Mocfi: A framework to mitigate control-flow attacks on smartphones. In: NDSS (2012)
4. Erlingsson, Ú., Abadi, M., Vrable, M., Budiu, M., Necula, G.C.: Xfi: Software guards for system address spaces. In: Proceedings of the 7th symposium on Operating systems design and implementation. pp. 75–88. USENIX Association (2006)
5. Haller, I., Göktaş, E., Athanasopoulos, E., Portokalidis, G., Bos, H.: Shrinkwrap: Vtable protection without loose ends. In: Proceedings of the 31st Annual Computer Security Applications Conference. pp. 341–350. ACM (2015)
6. Jang, D., Tatlock, Z., Lerner, S.: Safedispatch: Securing c++ virtual calls from memory corruption attacks. In: Proceedings of NDSS 2014 (2014)
7. Niu, B., Tan, G.: Monitor integrity protection with space efficiency and separate compilation. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 199–210. ACM (2013)
8. Team, T.C.: Undefinedbehaviorsanitizer, clang 3.9 documentation (2016), http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, [Online; accessed 9-July-2016]
9. Wang, Z., Jiang, X.: Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In: 2010 IEEE Symposium on Security and Privacy. pp. 380–395. IEEE (2010)
10. Zeng, B., Tan, G., Erlingsson, Ú.: Strato: A retargetable framework for low-level inlined-reference monitors. In: Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13). pp. 369–382 (2013)
11. Zeng, B., Tan, G., Morrisett, G.: Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 29–40. ACM (2011)

12. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical control flow integrity and randomization for binary executables. In: Security and Privacy (SP), 2013 IEEE Symposium on. pp. 559–573. IEEE (2013)
13. Zhang, M., Sekar, R.: Control flow integrity for cots binaries. In: Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13). pp. 337–352 (2013)