IS561 Project Report ParisDakarTech: Backward-edge Protection: Improvements on SafeStack and RETGUARD

Alexis Gacel, Ndeye Khady Ngom, Kai Lüke

ABSTRACT

The integrity of return addresses pushed to the stack is the oldest target of control flow attacks. No mainline compiler offers defenses for attacks that find the position of the return address and replace its content with a malicious target address. We study proposed solutions of the last years and try to overcome their security and compatibility problems. We present an accessible Clang compiler wrapper which offers shadow stacks or cryptographic return address integrity. Future x86 or ARM processors will enable the kernel to provide hardware shadow stacks (Intel) or fast pointer authentification (Qualcomm). But until then sensible applications without highest performance requirements can make use of our software solution.

1. INTRODUCTION

Many mitigation ideas have been implemented since the first steps were taken to defend against stack smashing attacks. Still, stack canaries are the established choice of C compilers and enabled by default. But besides buffer overflows there are many other vulnerabilities that offer read/write capabilities for an attacker. In general, canaries do not protect against a targeted overwrite of the return address if the canary is left intact. Also when the stack content and thus the canary is leaked the offer no protection.

Not all security-sensitive programs have performance as high goal and for others the advancement of processor speed should have compensated the impact of of more secure solutions. Looking at the two big free software compilers GCC and Clang it is surprising to see that only SafeStack [6] from CPI/CPS [16] got upstreamed into Clang. Safestack maintains an additional unsafe stack for buffers and objects with leakable pointers. At least OpenBSD patched their Clang version recently with RETGUARD [19], which XORs every return address with the RSP. Before we look at these two and their leak-related vulnerabilities in more detail, other defenses are presented.

After StackGuard introduced the concept of random canaries [3], similar strategies were developed like the ProPolice canary that finally landed in GCC, or XOR random canaries that XOR the return address with the canary. Pointguard was an approach to XOR all code pointers with a canary before they are saved to memory [2]. This attempt hurts compatibility and was therefore abandoned, in addition it is also vulnerable to leaks. StackShield implemented a shadow stack. The calling convention on SPARC allowed StackGhost to add a kernel handler for frame spills in order to introduce a XOR of the return address with a per-process key before it is saved to the stack [10].

The term Control Flow Integrity (CFI) is often used to refer to techniques for forward-edge protection i.e. integrity of indirect jumps and virtual function tables. But it has to be noted that CFI relies on available backward-edge protection which we discuss here [1]. The CFI version of Clang [7] does only mention a proposal in the design document to extend it further to backward-edge CFI for return statements [8].

The PaX team sells grsecurity RAP [12] which introduces CFI for user and kernelspace. This CFI is based on function types and also used for return target verification, where in addition an in-register XOR canary of the return address is kept. But in userspace this canary is stored on the stack and the protection is vulnerable to leaks the same way as other XOR-based solutions.

Cryptographically Enforced CFI (CCFI) [17] implements in contrast to all XOR-based solutions, a leak-resilient return address protection through AES-NI HMACs based on a secret key, the return address, and the RBP. Even though known-plaintext attacks are circumvented, a replay attack scenario is left where a previously valid address can be reinjected in an invalid context. As a custom LLVM pass it stores the key and AES variables in the `xmm` registers and breaks ABI compability.

Qualcomm announced an ARM 8.3 pointer authentification ISA extension that offers hardware implementations of HMACs for return address integrity or CFI [18]. For return address integrity the context for the HMAC is the RSP which also allows a replay attack. In comparison to CCFI the HMAC size is reduced to fit into the ~24 static bits of the userspace address pointers. The keys are stored in the processor and have to be managed by the kernel per process.

Microsoft included Return Flow Guard (RFG) [5] as shadow stack solution but discontinued this project because the API design was vulnerable to leaks and the region itself writable [15]. Instead co-

operation with Intel was started for Intel's hardware stack protection.

Intel announced its Control-flow Enforcement Technology (CET) [14] as transparent hardware shadow stack for `call/ret` instructions. The shadow stack of a process is set up by the kernel and protected through the MMU. In addition it includes legal-target markers for indirect jumps as simple CFI mechanism. Patches landed recently in Clang and GCC but neither kernel support nor hardware are available.

This project should implement and evaluate strong return address protection in software. This is considered meaningful for sensitive applications because the hardware-based solutions are just emerging at the horizon and will take years to arrive for the majority of users.

2. SAFESTACK AND RETGUARD

We first study the current problems of SafeStack and RETGUARD to see what properties a solution needs. SafeStack is designed to keep the original stack safe by static analysis to determine leakable pointers which should rather be used for the unsafe stack. Since the unsafe stack is isolated, buffer overflows there do not reach the return addresses but hit unmapped memory. The performance impact of moving buffers to a second memory region is small.

Instead of isolating memory regions, RETGUARD only needs an additional XOR instruction in the function prolog and epilog. It considers the RSP as key to be secret so that appropriate XORed values can not be forged by the attacker. But also reading the XORed stack content reveals the RSP if the target address is known (or the target address if the RSP is known) due to the nature of the XOR operation working in both directions.

To protect against an arbitrary write capability both defenses rely on ASLR and thus decline with pointer leakage. With SafeStack direct leaking of the stack position through variables is prevented, but information hiding is a complex problem specially when the position of the original stack is concerned. The stack position can be found through neglected pointers in libraries or the TCB, thread spraying or the constant offset from TLS for secondary threads [11]. On a simple implementation side channel attacks were possible, too [9].

In particular the relation of the TLS with secondary threads was a motivation for us to see what other offsets are constant. We found out that the unsafe stack has a constant offset to the stack of secondary threads. We were also able to leak all sensible pointers through a traditional format string attack because the x86_64 call convention for variable argument counts let us leak the stack contents like RBP and the return address after first the registers were leaked.

RETGUARD and SafeStack stay ABI compatible which is eases adoption. For RETGUARD there is a window of few processor cycles where the return address is unprotected in memory directly after the `call` or before the `ret` instruction. TOCTOU attacks that exploit this are feasible in theory but we do not consider them here.

Since RETGUARD like SafeStack relies on a hidden RSP value, a combination of RETGUARD and SafeStack offers no higher protection if this assumption is violated. For completeness we want to point out that RETGUARD does not incur a hurdle for ROP chains. Here an example with three gadgets and the malicious stack content being placed instead of the return address at RSP:

A: `pop rdi; pop rsi; xor [rsp],rsp; ret`
B: `call open; xor [rsp], rsp; ret`
C: `call read; …`

```
rsp+56:      C^(RSP+56)
rsp+40:      ptr:buf
rsp+40:      3
rsp+32:      A^(RSP+32)
rsp+24:      B^(RSP+24)
rsp+16       0
rsp+8:       ptr:"file"
rsp:         A^RSP
```

3. PROJECT THREAT MODEL AND REQUIREMENTS

Both SafeStack and RETGUARD do not protect against two consecutive format string attacks. The attack would first to leak the RBP (in SafeStack other *interesting* pointers could be used as well on secondary threads) to calculate the RSP as pointer to the return address, and also leak the return address as a code pointer to calculate the target address. Then the second format string attack overwrites the return address e.g. by using the RBP again with the `%n` argument to set up a pointer to the return address at the position where RBP pointed to, then later in the format string reference this as argument to continue with finally overwriting the return address with another `%n` argument.

Many other vulnerabilities give similar read/write primitives which might even be more convenient to use. We can abstract from all the specific attacks by requiring that any solution we propose for

return address integrity should mitigate against overwriting the return address after the stack contents have been leaked (stack contents can be stack and code pointers). Defenses can still be probabilistic which means that mitigation is not fully guaranteed.

Replay attacks need observation of a large execution path to inject old return addresses in a different context where they can not be distinguished from the valid addresses. If a solution does not prevent this, it is a drawback but overall still a huge improvement to the current state if this is the only vulnerable point.

Because we value compatibility we disregard the time window between the call instruction until the function prolog has finished as well from the start of the function epilog until the return address is executed. Reads and writes during this time are out of our threat model.

Defenses against the described leaks have to be more involved than a simple XOR and therefore we expect them to have a bigger impact on the runtime. Shadow stacks are easy solutions that fulfill the criteria but they are known to be slow. On the other hand this means that any cryptographic approach should be faster than shadow stacks, otherwise it makes no sense to use it because the properties are likely to be weaker than those of shadow stacks.

4. PROJECT PLAN AND RESULT

First we intended to accompany SafeStack with a RETGUARD-like XOR of the return address, but instead of the RSP using a secret value. As we realized that this is vulnerable to the described known-plaintext attack we studied stronger cryptograhic primitives for signatures, encryption and HMACs and also found out how CCFI and Qualcomm Pointer Authentification approached this.

The implementation also had unexpected challenges because a separate LLVM pass does not allow the replacement of machine instructions. Like RETGUARD we would have to modify LLVM with a new machine target pass. But there is not much documentation to find and the compile times allowed only four tries per hour. Therefore we decided to go with assembly instrumentation for the limited scope of this project. This also allowed us to produce more prototypes with different techniques for return address protection. Each prototype is a pass which operates on assembly artifacts of Clang.

Main building blocks for a compatible instrumentation of the function pro- and epilogs are a way to decide whether initialization code should be executed and a secure information storage. Thread local storage (TLS) variables, in C declared with the `__thread` prefix, can be declared with a initial value which allows detection if a new thread was started. For the safe storage place the TLS is not appropriate because it is writable in memory and is placed at the beginning of each secondary thread.

Since the System V 64 bit ABI does not guarantee regular registers to stay untouched we considered switching off AVX usage during compilation. This not only impacts performance but also hurts compability when linking. Luckily we found out that the old x87 floating point register stack (st0, st1, …) is not used by GCC or Clang and even can not be turned on with `-mfpmath=387`. We therefore consider these registers as safe because only our instrumentation code will use them. Binary ROP gadgets are irrelevant because they require that the control flow hijack already took place.

Next we will describe the assembly instrumentation framework, a RAP-like simple XOR pass, an userspace shadow stack pass, an in-kernel shadow stack pass and finally a pass with HMAC-based cryptographic protection.

4.1 INSTRUMENTATION FRAMEWORK

We designed a compiler wrapper script that will first produce assembly output for instrumentation and then continue with the final compilation to a linked binary. Because it should behave invoking the regular compiler some more tricks are involved but we hope to have most things covered. The instrumentation passes are kept in separate files and are specified as argument for the wrapper. An extra layer of complexity is that for supporting shared objects each pass needs to switch between the modes of addressing the TLS variables. Also, a pass should emit its global definitions only once.

```
# Set compiler for build scripts
# or manual invocation as $CC
export CC="$PWD/ccwrapper $PWD/pypass…"
```

For usage with a build system exporting the `CC` environment variable was enough in many cases, some build scripts might need adaptions. The test about how the compiler “reports undeclared, standard C functions” needs to be patched away for gzip. Another instance are `-O2` flags for optimization e.g. if the used instrumentation pass can only follow strict LIFO semantics the stack. But our framework may have other problems with optimized code, too.

The requirements for a pass are that it can be called with these arguments by the wrapper:

```
./passXYZ [--shared] [infile] outfile
```

A pass modifies an input assembly file in AT&T syntax and saves the result as output file, where input and output can be the same file. If no input file is given, the pass should emit global definitions which are only included once in the final compilation phase. Presence of the shared flag means that compilation in Clang takes place with the `shared` flag so that the emitted assembly can adapt addressing modes in terms of suffixes to the TLS:

```
# normal mode:
movq $1, %fs:var@tpoff
# vs.
# position independent mode:
leaq var@TLSGD(%rip), %rdi
callq __tls_get_addr@PLT
movq $1, (%rax)
```

Using Intel syntax turned out to be too compilicated and lead to inconsistencies due to implicit assumptions in the compilers that expect AT&T and when inline assembly is involved.

Our passes are written in Python and the main work takes place in the following line which relies on Clang's markers for function blocks.

```
inp.replace("retq\n", ret)
   .replace("# BB#0:\n", entry)
```

The variable `entry` contains instrumentation for the function prolog, `ret` for the epilog.

The simplest pass is just an empty bash script that does nothing. We also reimplemented RETGUARD as pass to be independed from LLVM builds which led to a high difference in runtime speed depending on the version that is used. The following sections describe all other passes.

4.2 SIMPLE XOR PROTECTION

We can improve RETGUARD with a secret key instead of the RSP. While it is still vulnerable to leaks of the XORed return address, a blind replacement after RSP inference is not possible anymore. How grsecurity RAP works in userspace is similar.

The function prolog checks the TLS variable to decide whether the key needs to be set up. Key setup is done in a helper function which uses the `getrandom` syscall and then stores the random key into the x87 register stack. At each function entry and exit the key is changed with a constant (or the RSP) in order to generate a temporary key for this function, as a weak mitigation of the most simple replay attacks. This updated key is stored back into the safe register. The prolog ends with cleaning the key from the memory and general register where it needed to go through in order to XOR the return address. Just before a return instruction is executed, the epilog undoes the XOR after retrieving the key from the x87 register stack and updating the key.

A better temporary key is needed because the current change per function frame is computable which allows to modify an observed XORd return address to be used in a different function frame. But since XOR is used the whole solution anyway leaks the current key if the plain return address is known. If a secure hash function would be used to generate a unique temporary key that can not be used to compute a temporary key of another function frame, then there is no point in using XOR anymore because a secure hash is already everything needed for a HMAC solution that is resilient against known plaintext attacks.

This approach does not fully meet our requirements but is very simple and fast. If a two step exploit of reading the XORed return address, knowning the real return address and then forging a new XOR is not a realistic threat model, this method could be recommended for performance reasons.

4.3 SHADOW STACK

Shadow stacks in the program memory have the challenge that they are writable. Resetting the mapping permissions in each function entry and exit is costly because it would involve system calls (but easy to add if needed). Our solution aquires a new memory region from the kernel at a random position through ASLR. The pointer to this stack is only stored in the safe register which is never leaked. While the idea is isolation, this mitigation still stays probabilistic to some extend.

The original return address is safed to the shadow stack in the prolog and retrieved from there in the epilog. The shadow stack pointer in the safe register is updated but one could also do a calculation based on the current RSP and skip the update. This would also solve the problem that strict LIFO semantics break some higher compiler optimization levels. Currently the return address on the stack is compared with the shadow stack copy, a mismatch lets the program abort. One could also decide to overwrite the original return address to hide a code pointer. Attack detection can still be done by observing changes in the overwritten value which could be just zeros or the RSP. Access to two memory regions is expected to have a performance impact but therefore a shadow stack is not vulnerable to the described known-plaintext and replay attacks.

4.4 IN-KERNEL SHADOW STACK

Unlike the userspace stack, our in-kernel shadow stack is based on strong isolation and can distrust the integrity of all process memory. The downside is a system call are needed at function entry and exit, such a context switch hurts the runtime in many ways.

Instead of accuiring a mmaped region, each thread now gets its own in-kernel data structure. We selected the message queue of the System V IPC primitives. Access is restricted to the current user privilige when the queue is set up. The queue identifier is kept in the safe register in order to protect it from corruption, so that redirection to a attacker controlled queue with permission `666` is not possible.

We had to abuse the message type to turn the FIFO semantics of the queue into LIFO by using the current RSP as message type. On retrieval the kernel needs to search through the queue which results in *O(n)* complexity depending on the number of call frames. The strict LIFO semantics could be loosened by clearing all elements of the same message type before a message is stored.

Currently forks do use the same queue because only new threads are detected. The mentioned problems can all be solved with a custom kernel module that needs no queue identifier by using the PID to offer a separate hashmap for each thread as storage for the return address. We did not look at the possibility to use eBPF for that purpose.

4.5 HMAC POINTER AUTHENTIFICATION

To protect against key leakage through XOR with a known return address we first looked into signature algorithms. The duality of RSA keys allows to 'encrypt' a message with the private key instead of the public key, so that if 'decrypting' it with the public key is possible the message is restored and its authenticity proven. Besides RSA there are other schemes with such a double property of authentification and content hiding; the Nyberg-Rueppel signature, Niederreiter encryption scheme and signcryption. We did not investigate these options and the performance they would offer – maybe there is some hidden gem but mostly it did not look promising compared with the HMAC approach that CCFI and Qualcomm Pointer Authentification took.

We decided to start with a simple HMAC algorithm to see how the rest of the instrumentation would look like. The HMAC could be stored somewhere else but we did not want to introduce a change in the memory layout of the function. Like Qualcomm we use the static bits of the address to store the HMAC. We take the highest 17 bit which because they are always zero for userspace addresses.

We based our first prototype HMAC on a (alleged) RC4 stream cipher. The initialization code sets up a secret key in the x87 register with random data from the kernel. At function entry a new HMAC is calculated as *hmac(key, RSP, return addr.)*. The values are concatenated as 192 bit input to the RC4 generator and by taking three bytes of its output we get the 17 bit HMAC. The RC4 code was written in C and compiled to assembly for further modification. Controlling the register usage was easier with a custom calling convention for this helper function.

At the function epilog the HMAC and return address are separated again and a new HMAC is calculated on the basis of the found return address. If both HMACs differ, execution is aborted.

The alleged RC4 algorithm runs two loops with 256 iterations which involve memory operations. The runtime is hurt too bad for practical usage. Qualcomm uses the QARMA cipher, CCFI uses AES-NI to generate the HMAC which also involves a lot of instructions but less memory accesses. Yet in our case we would need to backup all `xmm` registers since we want to stay compatible. Still this is the most promising approach for current hardware. The newest hardware could use Intel's SHA ISA extension for 17 bit trimmed hash of the secret key, return address and RSP. Of the many other secure hash functions we could use instead, BLAKE2 offers good benchmarking results. While a 17 bit hash does not sound much one has to keep in mind that the whole input is 192 bit. The attacker can control the return address but does not know the secret key in order to find a collision. Yet, the attacker can iterate the possible 64 bit keys and hope that if the same HMAC is found this is not due to a collision but a correct key – truncating the hash also has positive effects.

In the first prototype with RC4 we decided to leave the return address leakable, however since the key just occupies one of the x87 registers we could easily introduce a second key that is used for an additional XOR to hide the return address. This also introduces more unknown bits for the attacker in the HMAC.

Therefore we implemented our last variant with a fast non-cryptographic hash function [13] and the additional XOR. Under the assumption that a strong

cryptoanalysis is unlikely this is our most promising approach.

Replay attacks are still possible but including the RSP into the hash limits them to the same position on the stack. Introducing a per-function constant in the hash would further narrow this down to return only to legitimate call sites of the current function observed at this position in the stack. The practicability of replay attacks has not been studied, but it might be possible the same way CFI is vulnerable where a class of pointers has to be allowed because exact knowlege is not available at compile time.

5. EVALUATION

Both shadow stacks and pointer authentification are strong mitigation techniques that offer integrity for return addresses. Based on examples we will demonstrate the security and performance properties of the approaches.

5.1 SECURITY

The threat model included leakage of the stack contents before the return address is overwritten. We simulate this setting with a HTTP2-like service that processes serveral requests in the same connection. The client communication is handled in a new thread. We chose a format string vulnerability because it first leaks registers and then the stack for higher arguments. A client requests files by specifing their names and the format string vulnerability occurs when the server uses these filenames in its answer. The answer is always that there is no such file. Instead of having to use the format string attack when overwriting the return address we included a simple write primitive for brevity; it is available when the client requests the file “a” and then address and content can be entered.

The program leaks the buffer pointer to the unsafe stack from a register, and from the stack a pointer to a static string in the binary, the RBP and the return address. It also leaks `stdin` which could be used to get the address of `system()` but instead of needing ROP to set up the argument we included the function `success` that starts a shell, so that a simple code pointer offset is enough to determine the new return address. Here is a simple extract of the vulnerable function:

```
char buf[70];
char response[100];
char *header = "Not found: ";
int len = strlen(header);
strncpy(response, header, len);
fprintf(out,
"Request filename (empty to end):\nGET ");
while (fgets(&buf[0], 70, stream) != NULL
      && buf[0]!='\n') {
  snprintf(response+len, 70, buf);
  fwrite(response, 1, strlen(response), out);
  // insert easy write primitive here
  fprintf(out, "\nGET ");
}
```

An exploit can either use the unsafe stack pointer or the RBP to calculate the RSP value when the function returns because both have a constant offset. Exploiting RETGUARD is straightforward though XOR of the found return address with the RSP, applying the offset to the new target function and a final XOR with the RSP to gain a valid content to overwrite the return address.

Our simple XOR protection pass can also be exploited through the known-plaintext attack. Even though the return address is XORed with a secret, we can calculate the expected return address though the leaked static string pointer, and in reverse then even get the key through a XOR. This allows us to XOR the malicious target address with the key and use it to overwrite the return address.

The two shadow stacks and the RC4 or fast hash HMAC authentification can not be exploited because of strong isolation for the kernel shadow stack, for the userspace shadow stack because of the missing knowlege about the pointer which is kept in a safe register, and for the HMAC protection because of the missing knowlege of the key for the HMAC algorithm.

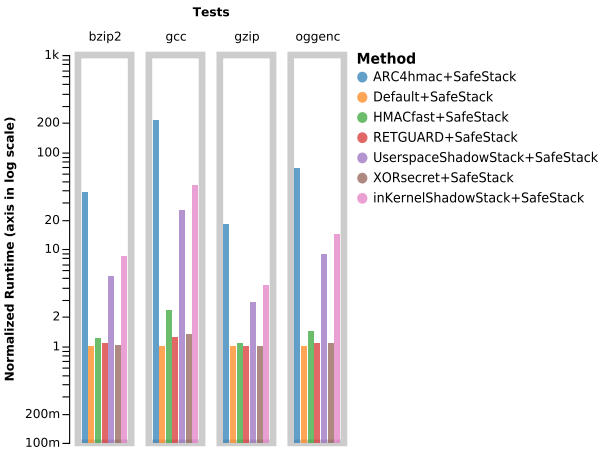
We provide the vulnerable program in `examples/vuln.c` and an automated exploit in `examples/vuln-exploit`.

5.2 PERFORMANCE

We measured the runtime of common programs to get an impression of the performance impact of the presented solutions. Microbencharks of the duration of prolog and epilog would also be possible in order to be independent from the number of call/return pairs in the programs. In literature the runtime increasement for shadow stacks varies a lot from 5% to 30% or even 50% but all these numbers have to be taken with a grain of salt [4]. We did no finetuning for our prototypes.

The simple XOR secret does not meet out security requirements but is almost as fast as RETGUARD or no additional protection – the varying runtime jitter is more significant than the slowdown. Out of our secure solutions the userspace shadow stack offers the best performance with a runtime of factor 3–25. The in-kernel shadow stack resulted in a runtime of factor 4–40. Our RC4-based HMAC turned out to be

very slow with a runtime of factor 38–190. The fast HMAC with code pointer hiding has a runtime factor between 1.2 and 2.3.



The programs were compiled without higher optimization. The input to bzip2 and gzip was the Calgary corpus, for oggenc we used a test input of the Opus codec website. The input for GCC was a large C file generated with `benchmark/input/genc`.

5.3 FUTURE WORK

The chosen general hash function has to be evaluated in terms of cryptoanalysis or be replaced with a lightweight secure hash function; the main goal is that the 64 bit key as unknown part of the algorithm input is not computable from observing the 17 bit truncated hash output. Also there are not many malicious target addresses for collisions.

We could also use the AES-NI instructions to compute an HMAC and see if this is faster than the userspace shadow stack. Due to ABI compability this will likely be slower than CCFI is able to keep all the `xmm` registers set up. We did not have hardware to test Intel's SHA instructions for HMAC calculation.

Replay attacks could be further made difficult by introducing a per-function constant into the HMAC.

Currently we instrument every function but it would be possible to adapt the `stack-protector-strong` behavior of GCC to reduce function coverage to those which are in need of protection.

Rerandomization of the key for young stack frames could be implemented because more than the half of the x87 registers are still free to backup the key for old stack frames. This is particularly useful for forked processes if the simple XOR secret pass is used.

All passes should be implemented as LLVM target machine pass or GCC plugin to improve compability with build scripts and compiler optimizations, as well as compile time compared to our assembly text replacement.

The userspace shadow stack needs to be further optimized. The kernel shadow stack is an interesting idea and could also be optimized or even generalized for secure information storage of other kinds than return addresses.

6. CONCLUSION

We presented attacks to SafeStack and RETGUARD and developed strong return address protection based on safe registers and shadow stacks or cryptographic authentification. The acceptable tradeoff depends on the application in question, but currently the fast HMAC is the recommended solution or the userspace shadow stack if one does fear a strong cryptoanalysis. The XOR secret pass offers weaker protection than our slower solutions but improves on the security of RETGUARD and SafeStack with minimal performance impact.

The availability of Intel's CET and Qualcomm's Pointer Authentification will likely obsolete our software solutions if implemented carefully. But even then there will be systems without these hardware features.

REFERENCES

[1] Nicolas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. 2015. Control-flow Bending: On the Effectiveness of Control-flow Integrity. In *Proceedings of the 24th USENIX Conference on Security Symposium* (SEC'15), 161–176. Retrieved from http://dl.acm.org/citation.cfm?id=2831143.2831154

[2] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. 2003. PointguardTM: Protecting Pointers from Buffer Overflow Vulnerabilities. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12* (SSYM'03), 7–7. Retrieved from http://dl.acm.org/citation.cfm?id=1251353.1251360

[3] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. 1998. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7* (SSYM'98), 5–5. Retrieved from http://dl.acm.org/citation.cfm?id=1267549.1267554

[4] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. 2015. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the*

10th ACM Symposium on Information, Computer and Communications Security (ASIA ccs '15), 555–566. DOI:https://doi.org/10.1145/2714576.2714635

[5] FlowerCode DannyWei Iywang. 2016. Return Flow Guard. Retrieved from http://xlab.tencent.com/en/2016/11/02/return-flow-guard/

[6] Clang 6 Documentation. 2017. SafeStack. Retrieved from https://clang.llvm.org/docs/SafeStack.html

[7] Clang 6 Documentation. 2017. Control Flow Integrity. Retrieved from https://clang.llvm.org/docs/ControlFlowIntegrity.html

[8] Clang CFI Design Documentation. 2017. Backward-edge CFI for return statements (RCFI). Retrieved from https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html

[9] Isaac Evans, Sam Fingeret, Julian Gonzalez, Ulziibayar Otgonbaatar, Tiffany Tang, Howard Shrobe, Stelios Sidiroglou-Douskos, Martin Rinard, and Hamed Okhravi. 2015. Missing the Point(Er): On the Effectiveness of Code Pointer Integrity. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy* (SP '15), 781–796. DOI:https://doi.org/10.1109/SP.2015.53

[10] Mike Frantzen and Mike Shuey. 2001. StackGhost: Hardware facilitated stack protection. In *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10* (SSYM'01). Retrieved from http://dl.acm.org/citation.cfm?id=1251327.1251332

[11] Enes Göktaş, Robert Gawlik, Benjamin Kollenda, Elias Athanasopoulos, Georgios Portokalidis, Cristiano Giuffrida, and Herbert Bos. 2016. Undermining Information Hiding (and What to Do about It). In *25th USENIX Security Symposium (USENIX Security 16)*, 105–119. Retrieved from https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/goktas

[12] grsecurity PaX Team. 2015. RAP: RIP ROP. Retrieved from https://pax.grsecurity.net/docs/PaX-Team-H2HC15-RAP-RIP-ROP.pdf

[13] Lockless Inc. 2017. Fast Hash Function. Retrieved from https://locklessinc.com/articles/fast_hash/

[14] Intel. 2017. Control-flow Enforcement Technology Preview Rev. 2. Retrieved from https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf

[15] Eyal Itkin. 2017. Bypassing Return Flow Guard. Retrieved from https://eyalitkin.wordpress.com/2017/08/18/bypassing-return-flow-guard-rfg/

[16] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-pointer Integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (OSDI'14), 147–163. Retrieved from http://dl.acm.org/citation.cfm?id=2685048.2685061

[17] Ali José Mashtizadeh, Andrea Bittau, David Mazières, and Dan Boneh. 2014. Cryptographically Enforced Control Flow Integrity. *CoRR* abs/1408.1451, (2014). Retrieved from http://arxiv.org/abs/1408.1451

[18] Qualcomm. 2017. Pointer Authentication on ARMv8.3. Retrieved from https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf

[19] Theo de Raadt. 2017. RETGUARD. Retrieved from https://lwn.net/Articles/732202/