2021 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)

Finding Runtime Usable Gadgets: On the Security of Return Address Authentication

Qizhen Xu[1,2], Zhijie Zhang[1,2], Lin Zhang[3], Liwei Chen[1,2,*], Gang Shi[1,2]
1. Institute of Information Engineering, Chinese Academy of Sciences
2. School of Cyber Security, University of Chinese Academy of Sciences
3. Center of Engineering and Construction Service, Ministry of Agriculture and Rural Affairs
Beijing, China
Email: {xuqizhen, zhangzhijie, chenliwei, shigang}@iie.ac.cn, ramona75@163.com

***Abstract*—Return address authentication mechanisms protect return addresses by calculating and checking their message authentication codes (MACs) at runtime. However, these works only provide empirical analysis on their security, and it is still unclear whether the attacker can bypass these defenses by launching reuse attacks.**

In this paper, we present a solution to quantitatively analysis the security of return address authentication mechanisms against reuse attacks. Our solution utilizes some libc functions that could leakage data from memory. First, we perform reaching definition analysis to identify the source of parameters of these functions. Then we infer how many MACs could be observed at runtime by modifying these parameters. Afterward, we select the gadgets that could be exploited by reusing these observed MACs. Finally, we stitch desired gadget to craft attacks. We evaluated our solution on 5 real-word applications and successfully crafted reuse attacks on 3 of them. We find that the larger an application is, the more libc functions and gadgets can be found and reused, and furthermore, the more likely the attack is successfully crafted.

***Index Terms*—Return address authentication, reuse attack, memory security**

I. Introduction

Due to the widespread existence of memory vulnerabilities within programs written in C/C++ language, attackers often corrupt the most common code pointer, return address, to hijack a program's control-flow, e.g. return-oriented programming [1], [2]. To counter such attacks, defenses either use isolation or encryption to protect return addresses. Shadow Stacks [3], [4] maintain a reference copy of return addresses, and ensure its integrity by either information hiding or using specialized hardware. However, information hiding is vulnerable to information leakage, and once the copy is exposed to the attacker, he can tamper the copy to bypass return address checking. A hardware-assisted scheme, Intel CET [5], provides a new page attribute for isolation, but employing a custom hardware mechanism incurs development costs.

Return address authentication mechanisms [6]–[8] use message authentication code (MAC) to protect the integrity of return addresses. It calculates and checks the MAC of each return address at runtime. It is easy to be deployed because cryptographic modules are available in most modern processors, and it just requires instrumenting MAC-related instructions on function prologues and epilogues during compilation. Unfortunately, the attacker can reuse previously observed return addresses and their MACs to pass the MAC check process. But no quantitative analysis has been conducted on CCFI's security against this reuse attack, so we bring out three questions and regard them as evaluation metrics:

RQ1: How many MACs can the attacker observe at runtime? A MAC is generated and pushed onto stack on a function call, and would be popped to be verified on function returns. If the attacker cannot read the MAC during the function execution, the MAC cannot be observed.

RQ2: How many gadgets can be found by reusing observed MACs? By reusing observed MACs, the attack can alter the control-flow to another return location. The code in the following control-flow graph may contain the gadgets that an attack requires.

RQ3: What threats can reusing observed MACs bring to the program? Given these observed MACs and gadget, what kind of threats can the attacker expose to the computer. In fact, reusing these MACs could be seen as a subset of code reuse attack.

We propose a solution to evaluate the practical security of return address authentication mechanisms. The solution consists of two techniques: SearchMAC and StitchMAC. SearchMAC tries to disclose MACs as many as possible at runtime, and StitchMAC utilizes these MACs to chain gadgets together to launch attacks.

SearchMAC first performs reaching-definition analysis on the parameters of libc functions, like *memcpy* and *strcpy*. We call them disclosure function. If their *dest* and *src* pointers are under attacker's control, their return addresses and MACs could be leaked at runtime, so as to their callers. Reusing the MACs, attacker can bypass the MAC checking process, and return to a call-preceded instruction. SearchMAC records these return points.

Then StitchMAC utilizes the payload frontend and block searching unit of BOPC [10], an automatic tool to assess control-flow integrity(CFI) defenses, to find desired basic blocks. Finally, StitchMAC directly injects observed return addresses to chain gadgets together to craft an attack.

We evaluated our solution on 5 popular applications. These

* Corresponding author

978-1-6654-3574-1/21/$31.00 ©2021 IEEE
DOI 10.1109/ISPA-BDCloud-SocialCom-SustainCom52081.2021.00059

applications contain dozens of controllable disclosure functions. But two of them have no observable MACs because they are quite small, while the others contain dozens and hundreds of observable MACs. Reusing these MACs, we can successfully launch attacks on the three applications. We also found that, the larger an application is, the more reusable gadget it contains, and the more likely an attack is successfully crafted.

In summary, we make the following contributions:

- We propose three evaluation metrics to quantitatively analysis the practical security of return address authentication mechanisms.
- We design an analysis technique to discover observable MACs, which uses reaching definition analysis and path trace.
- We design an reuse attack generation approach based on the observed MACs.
- We implemented our techniques in Python, which utilizes the reaching definition engine in angr and some modules of BOPC.
- We evaluated our techniques on 5 real-world application, and successfully generate exploits on 3 of them.

Paper Organization. The rest of the paper is organized as follows. Section II provides the background of return address authentication mechanisms, the evaluation metrics of CFI, and exploit generation tools. Section III provides the threat model and assumptions. Section IV presents the design of our solution. Section V provides our implementation details. Section VI shows the experiment results. Section VII discusses its limitation and related works. Section VIII concludes this paper.

II. Background and Related Work

A. Return Address Authentication

Cryptographically control-flow integrity (CCFI) [6] firstly proposed the idea of using MAC to authenticate return addresses. In the function prologue, it calculates the MAC of return address, and pushes the MAC onto the stack. Before function returns, it pops the return address, recomputes its MAC and compares it with the MAC popped from stack. If the MAC does not match, it crash the program. A similar mechanism has been proposed on ARM architecture [7]. These return address authentication mechanisms can be an alternative to Shadow Stack, because cryptographic modules are widely deployed in modern processors, and attackers cannot generate MACs without the secret key.

However, despite that the attacker cannot directly overwrite the MACs, he can reuse observed MACs and corresponding return addresses to hijack the control-flow. CCFI tries to limit this reuse attack by adding the frame address into the input of MAC calculation, but it only provides theoretical analysis to illustrate their security against reuse attack. The real attack surface still remains questionable.

B. The Evaluation Metrics of CFI and Authentication

CFI [11] mitigates control-flow hijacking attacks by limiting the targets of indirect control transfer (ICT) instructions to a pre-computed valid set. Previous measurements for CFI usually focus on the average indirect target reduction(AIR) or gadget reduction [12].

The AIR metric [12] measures the relative reduction in the average number of valid targets for all indirect branch instructions that a CFI scheme provides: without CFI protection, an indirect branch could target any instruction in the program; CFI limits this to a set of valid targets. The gadget reduction metric measures the relative reduction in the number of gadgets that can be found at locations that are valid targets for an indirect branch instruction.

However, control-flow bending (CFB) [13] demonstrates that these metrics are too coarse-grained. Even with an AIR of 99%, an application that contains 10MB of executable memory still has 100,000 potential targets, which is much more than the number that a successful ROP attack usually requires.

Inspired by these works, we think it is the absolute number not the reduction percentage that matters. Therefore, we propose three evaluation metrics for return address authentication mechanisms: observable MACs, reusable gadgets and how useful they are to an attacker. The observable MACs are similar to the legal targets in CFI, but that they are generated and disappeared at runtime, which means if they cannot be leaked during life cycle, they cannot be reused. The reusable gadgets are similar to the available gadgets in CFI.

C. Exploit Generation Tool

Since Shacham *et al*. [2] proposed ROP technology in 2007, many researchers try to develop automatic exploit generation tool. Schwartz *et al*. [14] showed a highly reliable automatically generating ROP chains method named Q. Q is by far the most classical technique and accepted widely. But it does not consider CFI or authentication protection.

More recent work BOPC [10] tries to find usable gadget to launch attacks under CFI constraint. BOPC provides two useful tools, the SPloit Language (SPL) and Block Constraint Summaries (BCS). SPL allows analysts to express exploit payloads in a compact high-level language that is independent of target programs. BCS enables basic blocks to be employed in a variety of different ways by abstracting each blocks from individual instructions. That is, a basic block that modifies a register in a manner that may fulfill an SPL statement may be used as a gadget in latter exploit generation process.

III. Threat Model and Assumptions

In our threat model, we assume a binary is protected with return address authentication, and the kernel does not save user-level registers that contain the key during context switches in user accessible memory. Therefore, the attacker cannot directly modify a turn address and its MAC. Besides, we do not consider brute-force attack.

But we assume the binary contains a known arbitrary memory write corruption vulnerability, and the vulnerability can be triggered multiple times. This is realistic because many web servers often create threads to handle requests, and if a request handle contains such a vulnerability, the attacker can repeat arbitrary memory write multiple times.

We assume an attacker who tries to perform MAC reuse attack can write all writable memory by exploiting the vulnerability. We call this *arbitrary memory write primitive* (AWP). We also assume that there exists an entry point that the program reaches naturally after all AWPs complete. This assumption is the same with that in BOPC. We set the entry point near the AWP in CFG.

Since other defenses such as ASLR can be bypassed by information leakage, they are orthogonal to our work.

IV. Design

In this section, we describe how SearchMAC searches for runtime available MACs, and how StitchMAC chains desired gadgets together.

A. Available MACs Recognition

In our threat model, AWP can be triggered multiple times, so attackers can use the AWP to control the source and destination of memory read operations to leak out some return addresses and their MACs.

1) disclosure function: SearchMAC makes use of *disclosure function* to leak out return addresses and MACs. A disclosure function is one that can read its own return address when given arguments supplied by the attacker. If we can find a disclosure function that will be called later and use the vulnerability to control its arguments, we can make it read its own return address to the attacker. Of course, it can read its caller function's return address, too.

Any function that calls a disclosure function is itself a disclosure function: the disclosure function can read the return address of its caller(or higher on the call chain).

Many disclosure functions can be found in libc, e.g., *memcpy*, *strcpy*, *strcat*, and so on. Take *memcpy()* for example. If we control the *src* and *dest* arguments to *memcpy()*, we can point the source buffer to the return address and MAC, the target buffer to an attack-controlled region. The length to its word size is unnecessary because AWP can be triggered multiple times. When *memcpy()* is invoked, it will read its return address and MAC to the attacker-control location.

Other functions that contain buffer read overflow can be disclosure functions, too. But finding such functions is more difficult. We choose libc functions because they are widely used in applications. What's more, they are usually the leaf functions, which can leak more return addresses.

2) controllable disclosure functions: The *dest* and *src* parameters of disclosure functions may be constants or variables. Constants, such as array address, cannot be modified, so they cannot be exploited by the attacker. Variables can be modified, including local variable and global variable. If the *dest* and *src* parameters are both variables, the disclosure function is controllable.

In function calling conventions, the *dest* and *src* parameters of disclosure functions usually correspond to the *rdi* and *rsi* register. SearchMAC performs reaching definition analysis on the call point about the two registers. In binary analysis, they may come from another register or a memory location. Then SearchMAC walks back in the program dependence graph repeatedly to find whether their definitions are a constant or a local variable. If a parameter comes from a return value of a callee function, SearchMAC does not go into the callee function and treats the definition as a local variable. If either definitions of the parameters is a constant, the function cannot be controlled.

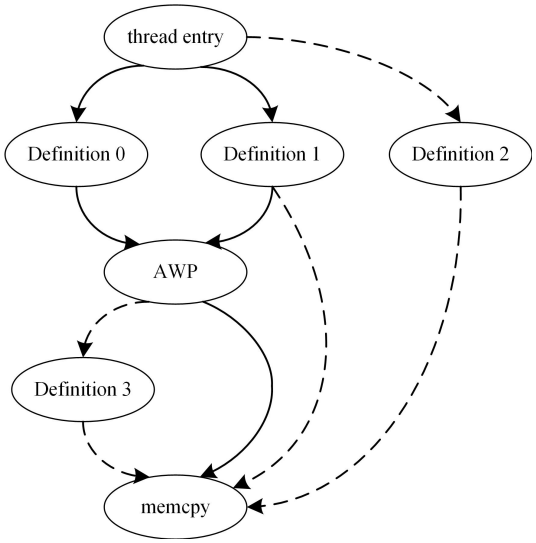

Fig. 1. Reachable disclosure functions. Arrows represent control-flow path. Solid arrows indicate the definition can pass the AWP node and finally reaches the disclosure function, while dashed arrows cannot.

If the definition may come from function parameters, SearchMAC performs inter-procedural analysis repeatedly. Reaching definition analysis is repeated on the caller function that calls disclosure function, until it finds the definition is constant or variable. After analysis, SearchMAC gets the definition address of parameters of disclosure functions.

3) reachable disclosure function: After finding the disclosure functions whose parameters are controllable, SearchMAC judges whether the attack can corrupt the parameters or not. If there is a path in the CFG that starts from the definition point, passes the AWP node, and finally reaches the disclosure function, the function can be controlled by attack. If the AWP is prior to the definition, or the AWP cannot reach the disclosure function, the function cannot be controlled.

Take Fig. 1 as an example. Suppose the *src* parameter of *memcpy()* has four definitions. There are paths that start from definition 0 and definition 1, pass the function where AWP can be triggered, and finally reaches the *memcpy()*. The two definitions are under the control of AWP. Definition 2 does not pass the AWP node, and definition 3 occurs after AWP node, so they cannot be tampered by AWP node.

4) observable MACs: After identifying controllable disclosure functions, the return addresses in the definition path that passes the AWP node can be observed, e.g., solid arrows in Fig. 1. Step further, the return addresses in all paths that reach controllable definitions can be observed. However, in web applications, it is not easy to trace back from definitions to the program entry, because these applications usually create threads

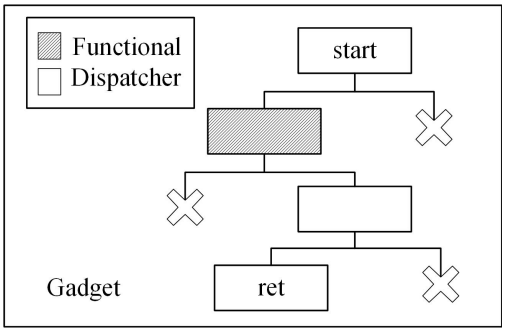

Fig. 2. Gadget structure. The functional blocks executes an SPL statement, the dispatcher blocks do not affect the execution of functional blocks, and the crosses indicate that the execution would be clobbered.

by indirect call and binary analysis cannot find the path from the thread entry to main loop. Therefore, SearchMAC tries to find as many paths to the thread entry as possible.

B. Reusing Observed MACs

After observing available MACs, the attacker can reuse these MACs to perform ROP attacks. However, unlike traditional ROP attacks, reusing MACs is more difficult. On one hand, crafting an exploit requires a lot of manual effort, let alone we want to explore how many kinds of threats can reuse attacks bring to the program. On the other hand, reusing a MAC can only return to a call-preceded basic block, thus it is more likely to fail. For convenience, we call this block *start node*.

1) payload and required basic blocks: To solve the two problems, StitchMAC first makes use of the SPloit Language (SPL) of BOPC. BOPC has provided several SPL payloads, which can be translated into several statements.

To find a sequence of blocks that implement each statement in the SPL payload, StitchMAC then leverages the Block Constraint Summaries (BCS) of BOPC to abstract all available blocks. These blocks lie in the paths that start from the *start node* to the block that the function returns, which is called *return node*.

After abstraction, StitchMAC gets the possible impacts that each block would have on SPL statements. Note that a basic block may satisfy multiple statements of SPL payload.

2) finding gadgets: We refer to a block that satisfies an SPL statement as *functional block*, a block that has no affect on the execution of functional blocks as *dispatcher block*, and the others as *clobbering block*.

From the *start node* to the *return node*, if there is a path that consists of one or more functional blocks and dispatcher blocks, we call this a *gadget*, as shown in Fig. 2.

After finding such a gadget (or several ones), StitchMAC leverages *concolic execution* [15] (symbolic execution along a given path). Along the way, it collects the required constraints that lead to the execution path.

3) stitching gadgets: If gadgets that satisfy each SPL statement have been found, StitchMAC chains them by observed return addresses and MACs. However, dispatcher blocks between different gadgets might still clobber each other, making the execution fail.

StitchMAC starts with an empty state, and launches concolic execution from the first gadget that satisfies the first SPL statement, then goes on until the last one. During the execution, if a gadget clobbers the previous execution, StitchMAC tries another gadget that satisfies the same SPL statement. If all gadgets clobber previous execution, StitchMAC goes back to the previous SPL statement, and tries another gadget.

After the last gadget finishes, StitchMAC checks whether the constraints have a satisfying assignment. These constraints will be concretized and translated into a memory layout that will be initialized through AWP in the target binary.

V. Implementation

In this section, we provide the implementation details of our solution. SearchMAC and StitchMAC are implemented in Python, and uses angr [16] for binary analysis.

A. Overview

Fig. 3 shows the overview of our implementation. SearchMAC performs reaching definition analysis to search for observable MACs, then it finds all available blocks to generate a reduced CFG and passes the reduced CFG to StitchMAC. StitchMAC first makes use of the SPL frontend in BOPC to generate various payloads, and leverages the BCS units to abstract the reduced CFG. Afterward StitchMAC searches for gadgets that satisfy SPL statements. Using concolic execution, StitchMAC simulates whether these gadgets can be chained together to form the final exploit. If they can, StitchMAC outputs a sequence of (address, value, size) tuples that describe how the memory should be modified to execute the payload generated from the SPL Frontend.

B. Reaching Definition Analysis

SearchMAC uses angr's reaching definition engine to search for the definitions of the parameters of disclosure functions, as listed in Algorithm 1.

Given a call instruction that invokes a disclosure function, the *ReachingDefinitions* returns its definitions of register, memory and so on. According to the calling convention of *memcpy()*, the *rdi* register corresponds to the *dest* pointer, and the *rsi* register corresponds to the *src* pointer. We can easily get the definition of rdi and rsi.

But usually they come from another register, and its value is still *Undefined*. So SearchMAC walks back in the *program dependence graph* to find its predecessor, until it reaches the concrete definition.

We divide the concrete definitions into three types. The first one is constant, the second is local variable, the last one is function parameter. Now that currently the *ReachingDefinitions* cannot identify a global variable accurately, we treat it as a constant. This would reduce the number of available MACs, but it also minimizes false positives and ensures the correctness of the result.

A definition is a set that contains at least three elements: atom, code location and dataset, optionally for tag. If a data

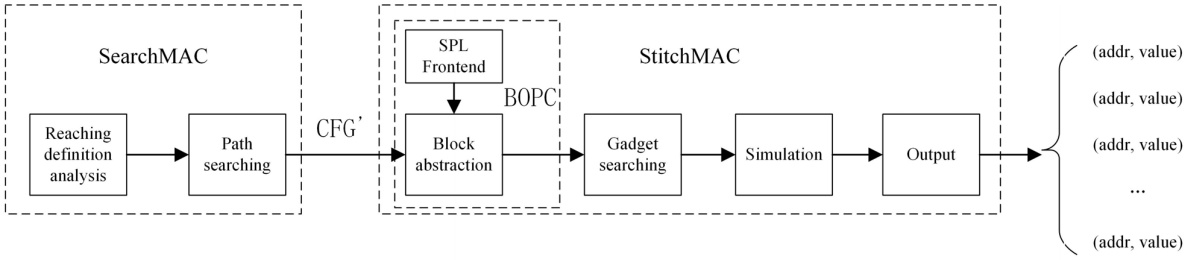

Fig. 3. Overview of the reuse attack generation.

Algorithm 1: Reaching definition analysis

```
Data: call memcpy() point
Result: definitions of the src parameter of memcpy()
def_rsi = angr.ReachingDefinitions.rsi;
check_queue.add(def_rsi);
addr = set();
while len(check_queue) != 0 do
    undef = False;
    def = check_queue.pop();
    seen_defs.add(def);
    if def.tag == LocalVariable then
        addr.add(def.addr);
    else if def.type == Undefined then
        undef = True;
    end
    if undef == True then
        for pred in dependence graph do
            if pred is parameter then
                para = angr.ReachingDefinitions.pred;
                repeat this algothism;
            else if pred is return_value then
                addr.add(pred.addr);
            else
                check_queue.add(pred);
            end
        end
    end
end
```

in the dataset is an integer, SearchMAC treats it as a constant. If the data has a LocalVariableTag, SearchMAC treats it as a local variable, and records the address of the definition from the code location object.

If the data type is still *Undefined*, SearchMAC walks back in the dependence graph to find its predecessors. If the predecessor comes from a previous statement, SearchMAC adds it to check queue. If it comes from a return value, either from a libc function or callee function, SearchMAC does not trace into the callee function and just treats it as a local variable. If it comes from an external function, which means it is a function parameter, SearchMAC replaces the *memcpy()* with its caller function, and repeats the process. If both *dest* and *src* are variables, the disclosure function is controllable.

C. Available MACs and Basic Blocks

For controllable disclosure functions, SearchMAC judges whether the attacker can corrupt their definitions. Since their *dest* and *src* parameters may come from different functions, SearchMAC chooses one of definitions that the other will pass in CFG as their collective definition.

SearchMAC uses a backward depth-first search algorithm, which starts from a disclosure function and goes back in the CFG until its definition node. If it encounters the AWP node before the definition node, the function is reachable.

For reachable functions, in theory, they can leak out all return addresses and MACs along the path. Since MACs are generated at runtime, but we did not run binaries so many times, therefore, SearchMAC only records return addresses for simplification. Note that this should be equivalent to leak out both MACs and return addresses.

After getting these return addresses, the basic blocks along the paths that from *start node* to *return node* are marked as available ones, which make up a reduced CFG and are transmitted to StitchMAC.

D. Candidate Gadgets

StitchMAC makes use of BOPC to abstract the reduced CFG and leverages SPL frontend to generate various payloads. These payloads consist of several statements. If a block that satisfies one or more SPL statements, StitchMAC searches for paths that include the block.

StitchMAC leverages concolic execution to simulate these paths until it finds one that consists of functional and dispatcher blocks. Blocks along this path make up a candidate gadget for an SPL statement.

If there is no functional block or candidate gadget for an SPL statement, the exploit generation fails.

E. Synthesizing Exploits

An SPL statement may have one or more candidate gadgets. StitchMAC initializes an empty state, and launches concolic execution again to simulate the whole gadget chain.

Fig. 4 shows the simulation process. If a gadget clobbers the SPL state during simulation, StitchMAC tries another candidate gadget that satisfies the same SPL statement. If all gadgets fail, StitchMAC goes back to the previous SPL statement, and tries another gadget.

TABLE I
AVAILABLE BASIC BLOCKS AND OBSERVABLE MACS.

program	vulnerability	all DFs	reachable DFs	observable MACs	available blocks
nginx1	CVE-2013-2028 [17]	172	75	203	1238
opensshd	CVE-2001-0144 [18]	37	22	0	0
proftpd	CVE-2006-5815 [19]	35	21	41	294
sudo	CVE-2012-0809 [20]	37	15	2	8
wuftpd	CVE-2000-0573 [21]	138	53	43	470

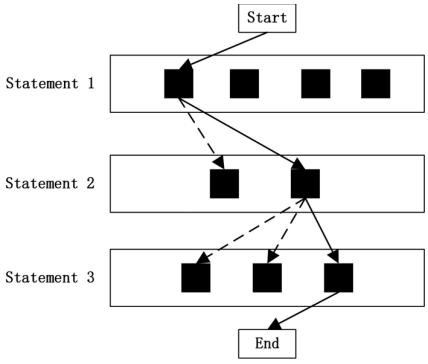

Fig. 4. The simulation process for the whole gadget chain. Black boxes in the rectangles represent candidate gadgets for SPL statements. Solid arrows indicate successful simulation, while dashed arrows indicate the next gadget would clobber previous simulation.

After the last gadget finishes, StitchMAC outputs a sequence of tuples (address, value, size) that describe how the memory should be initialized by AWPs.

VI. EVALUATION

We evaluate our solution on a set of 5 applications with known memory corruption CVEs, as listed in the second column of Table I, which fulfill our AWP requirement. Our evaluation aims to answer the following three questions brought in Section I:

- RQ1: How many MACs can be observed at runtime?
- RQ2: How many gadgets are available by reusing these MACs?
- RQ3: What threats can reusing observed MACs bring to the program?

A. The Number of Observable MACs and Available Blocks

For RQ1 and RQ2, we list the results in Table I, where DF is the abbreviation of disclosure function. The third column shows the total number of disclosure functions in binaries. The fourth column shows the number of controllable disclosure functions. The last two columns show the number of observable MACs and available blocks.

From Table I, we can see libc functions are widely used in popular applications, and nginx and wuftpd even contain more than one hundred disclosure functions. Our reaching definition analysis identifies dozens of reachable disclosure functions in these applications. However, the AWP node cannot reach any disclosure functions in opensshd, so there is no observable MACs in opensshd. Besides, there are only two observable MACs in sudo, which is too little to for exploitation. So we discard the two applications in the following evaluation.

The two cases are in our expectation that our solution does not always succeed. The opensshd and sudo application are quite small (441kb and 179kb), and their vulnerabilities may lie in some corners, so the AWP cannot control disclosure functions.

While the other three contain enough observable MACs. Among them, nginx contains over 200 observable MACs, which means the attacker should have enough candidate gadgets. Note that the binary size of nginx is only 3MB.

By reusing these MACs, proftpd and wuftpd have hundreds of available basic blocks, and nginx has over one thousand available blocks.

TABLE II
SEMANTIC MATCHING OF SPL STATEMENTS TO BASIC BLOCKS.

Statement	Example
Register Assignment	movzx rax,7h
Register Modification	dec rsi
Memory Read	mov rax, [rbx]
Memory Write	mov [rax], rbx
Call	call execve
Conditional Jump	test rax, rax jnz Loop

TABLE III
THE NUMBER OF FUNCTIONAL BLOCKS IN VULNERABLE APPLICATIONS

program	functional blocks						
	regset	regmod	memrd	memwr	call	cond	total
nginx1	1348	36	51	19	3	172	1629
proftpd	435	5	17	0	1	23	481
wuftpd	788	0	0	0	0	43	831

B. The Number of Functional Blocks

Before answering the RQ3, we first count the number of functional blocks. Since opensshd and sudo are discarded, StitchMAC abstracts the available blocks in the other three applications to find semantically matched SPL statements, as listed in Table II.

The number of identified functional blocks is listed in Table III. Note that the total number of functional blocks is larger than available blocks, because a block contains several assembly instructions, which may match different SPL statements.

TABLE IV
RESULTS OF RUNNING DIFFERENT SPL PAYLOADS

program	SPL payload									
	regset4	regref4	regset5	regref5	regmod	memrd	memwr	print	if-else	execve
nginx1	✓	✓	✓	✓	✓	✓	$\times_2$	$\times_3$	✓	✓
proftpd	✓	✓	$\times_3$	✓	$\times_1$	$\times_2$	$\times_1$	$\times_1$	$\times_2$	$\times_1$
wuftpd	✓	✓	✓	✓	$\times_1$	$\times_1$	$\times_1$	$\times_1$	✓	$\times_1$

C. Exploit Generation

For the RQ3, Table IV shows the feasibility of executing 10 SPL payloads for the three vulnerable applications. These payloads show what kinds of threats can an attacker bring to the system.

The *regset4* payload initializes four registers with arbitrary values. The *regref4* payload initializes four registers with pointers to arbitrary memory. The *regmod* payload initializes a register with an arbitrary value and modifies it. The *memrd* payload reads data from arbitrary memory. The *memwr* payload writes data to arbitrary memory. The *print* payload prints a message to stdout. The *execve* payload spawns a shell through the execve call. The *if-else* payload is an if-else condition based on a register comparison.

A checkmark means that the SPL payload was successfully executed on the target binary, while a cross indicates a failure. The subscript of a cross denotes the type of failure. Subscript 1 indicates there is no enough functional blocks, subscript 2 indicates there is no enough MACs, and subscript 3 indicates candidate gadgets clobber each other, making the exploitation fail.

For nginx, its success rate is obviously higher than others, because it contains the most functional blocks and MACs. For proftpd, it lacks *memwr* and *call* blocks, so related payloads all failed with failure type 1. For wuftpd, its functional blocks are concentrated in *regset* block, so it successfully generates *regset* and *regref* related payload, but the remaining payloads almost all failed.

The result shows that if an application contains more observable MACs and available blocks, it would have higher success rates to generate an exploit.

VII. DISCUSSION AND FUTURE WORK

A. Limitation of Binary Analysis

The reaching definition engine in angr cannot guarantee absolutely accurate results. Besides, address aliasing is another weakness on binary analysis. Therefore, SearchMAC just tries to eliminate false positive in definition results by ignoring some situations like global variable. In this way, SearchMAC gets a subset of the accurate result. Fortunately, experiment shows that the subset already contains enough gadgets for attackers to exploit.

B. ROP, BOP and MAC Reuse Attack

Traditional ROP attacks reuse code snippets that ended with a ret instruction. These gadgets usually contain no more than 10 instructions, so the rate of launching a successful attack is very high. While reusing MACs has to start with a call-preceded instruction, and the gadgets can be very long, leading to higher rate of failure.

BOP attacks assesses the security of a binary hardened with CFI and shadow stack defenses. Shadow stack is different with MAC authentication. With Shadow stack protection, a ret instruction can only return to the location that preceded by the latest call instruction. While in MAC authentication, a ret instruction can jump to any observed return addresses whose MACs are leaked.

C. CCFI's Limitation on Reuse Attack

CCFI limits reuse attack by adding the frame address into the input of MAC calculation. Since the frame address is stored on the stack, nearby the return address and MAC, an attackers can also leak it out. But in our current implementation, we did not really leak the frame address and MACs, because we did not run binaries so many times and our work aims to provide guidance for real exploitation.

D. Forward-edge MAC Reuse Attack

CCFI encrypts not only return address, but also function pointer, method pointer and vtable pointer. If these pointers can be observed, more fine-grained reuse attack can be crafted, increasing the success rate. However, CCFI adds pointer type information to the MAC calculation, which has been discarded in binary, so angr cannot analysis these elements. This work should be done by source code analysis.

VIII. CONCLUSION

The paper presented two techniques to quantitatively evaluate the practical security of return address authentication mechanisms against reuse attacks. The first technique collects observable MACs and records reusable basic blocks. The other one simulates these basic blocks and makes use of them to generate exploits.

Experiment results on 5 popular applications showed that reuse attacks might be successfully crafted by our techniques. Small binaries would have higher failure rate, because they may have not enough disclosure functions or its AWP node may reach zero disclosure functions. The large an application is, the more disclosure functions and gadgets are likely to be found and reused, and the higher success rate the exploit generation would have. We come to a conclusion that return address authentication is not secure enough against reuse attacks.

IX. Acknowledgment

We thank the anonymous reviewers for their valuable comments. This work is partially supported by the National Natural Science Foundation of China (No. 61602469), and the Fundamental theory and cutting edge technology Research Program of Institute of Information Engineering, CAS(Grant No. Y7Z0411105).

References

[1] M. Tran, M. Etheridge, T. K. Bletsch, X. Jiang, V. W. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Recent Advances in Intrusion Detection - 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings*, ser. Lecture Notes in Computer Science, R. Sommer, D. Balzarotti, and G. Maier, Eds., vol. 6961. Springer, 2011, pp. 121–141. [Online]. Available: https://doi.org/10.1007/978-3-642-23644-0_7

[2] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28-31, 2007*, P. Ning, S. D. C. di Vimercati, and P. F. Syverson, Eds. ACM, 2007, pp. 552–561. [Online]. Available: https://doi.org/10.1145/1315245.1315313

[3] T. H. Y. Dang, P. Maniatis, and D. A. Wagner, "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, Singapore, April 14-17, 2015*, F. Bao, S. Miller, J. Zhou, and G. Ahn, Eds. ACM, 2015, pp. 555–566. [Online]. Available: https://doi.org/10.1145/2714576.2714635

[4] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," in *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2019, pp. 985–999. [Online]. Available: https://doi.org/10.1109/SP.2019.00076

[5] Intel, "Control-flow enforcement technology specification, revision 3.0," 2019.

[6] A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI: cryptographically enforced control flow integrity," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, I. Ray, N. Li, and C. Kruegel, Eds. ACM, 2015, pp. 941–951. [Online]. Available: https://doi.org/10.1145/2810103.2813676

[7] H. Liljestrand, T. Nyman, K. Wang, C. C. Perez, J. Ekberg, and N. Asokan, "PAC it up: Towards pointer integrity using ARM pointer authentication," in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. USENIX Association, 2019, pp. 177–194.

[8] J. Zhang, R. Hou, J. Fan, K. Liu, L. Zhang, and S. A. McKee, "Raguard: A hardware based mechanism for backward-edge control-flow integrity," in *Proceedings of the Computing Frontiers Conference, CF'17, Siena, Italy, May 15-17, 2017*. ACM, 2017, pp. 27–34. [Online]. Available: https://doi.org/10.1145/3075564.3075570

[9] Z. Zhang, J. Wu, J. Deng, and M. Qiu, "Jamming ACK attack to wireless networks and a mitigation approach," in *Proceedings of the Global Communications Conference, 2008. GLOBECOM 2008, New Orleans, LA, USA, 30 November - 4 December 2008*. IEEE, 2008, pp. 4966–4970. [Online]. Available: https://doi.org/10.1109/GLOCOM.2008.ECP.950

[10] K. K. Ispoglou, B. AlBassam, T. Jaeger, and M. Payer, "Block oriented programming: Automating data-only attacks," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*, D. Lie, M. Mannan, M. Backes, and X. Wang, Eds. ACM, 2018, pp. 1868–1882. [Online]. Available: https://doi.org/10.1145/3243734.3243739

[11] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security, CCS 2005, Alexandria, VA, USA, November 7-11, 2005*, V. Atluri, C. A. Meadows, and A. Juels, Eds. ACM, 2005, pp. 340–353. [Online]. Available: https://doi.org/10.1145/1102120.1102165

[12] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, S. T. King, Ed. USENIX Association, 2013, pp. 337–352. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/Zhang

[13] N. Carlini, A. Barresi, M. Payer, D. A. Wagner, and T. R. Gross, "Control-flow bending: On the effectiveness of control-flow integrity," in *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015*, J. Jung and T. Holz, Eds. USENIX Association, 2015, pp. 161–176. [Online]. Available: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/carlini

[14] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: generalizing return-oriented programming to RISC," in *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, P. Ning, P. F. Syverson, and S. Jha, Eds. ACM, 2008, pp. 27–38. [Online]. Available: https://doi.org/10.1145/1455770.1455776

[15] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, M. Wermelinger and H. C. Gall, Eds. ACM, 2005, pp. 263–272. [Online]. Available: https://doi.org/10.1145/1081706.1081750

[16] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/firmalice-automatic-detection-authentication-bypass-vulnerabilities-binary-firmware

[17] CVEnginx2013, "Cve-2013-2028: Nginx http server chunked encoding buffer overflow 1.4.0." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2028

[18] CVEopenssh2001, "Cve-2001-0144: Integer overflow in openssh 1.2.27." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2001-0144.

[19] C. 2006, "Cve-2006-5815: Stack buffer overflow in proftpd 1.3.0." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5815.

[20] CVEsudo2012, "Cve-2012-0809: Format string vulnerability in sudo 1.8.3." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-2299.

[21] CVEwuftpd2001, "Cve-2000-0573: Format string vulnerability in wu-ftpd 2.6.0." [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0573