Physical Unclonable Functions-based Linear Encryption against Code Reuse Attacks

Pengfei Qiu, Yongqiang Lyu
Research Institute of Information Technology & TNList
Tsinghua University
Beijing, China
{qp15;lyuq}@mails.tsinghua.edu.cn

Di Zhai, Dongsheng Wang
Research Institute of Information Technology & TNList
Tsinghua University
Beijing, China
wds@mails.tsinghua.edu.cn

Jiliang Zhang*, Xingwei Wang
Department of Information Security, Software College
Northeastern University
Shenyang, China
zhangjl@swc.neu.edu.cn

Gang Qu
Electrical and Computer Engineering
Department and Institute for Systems Research
University of Maryland, College Park
Maryland, USA
gangu@umd.edu

ABSTRACT
Recently, code reuse attacks (CRAs) have emerged as a new class of ingenious security threats. Attackers can utilize CRAs to hijack the control flow of programs to perform malicious actions without injecting any codes. Existing defenses against CRAs often incur high memory and performance overheads or require extending the existing processors’ instruction set architectures (ISAs). To tackle these issues, we propose a hardware-based control flow integrity (CFI) that employs physical unclonable functions (PUF)-based linear encryption architecture (LEA) to protect against CRAs with negligible hardware extending and run time overheads. The proposed method can protect ret and indirect jmp instructions from return oriented programming (ROP) and jump oriented programming (JOP) without any additional software manipulations and extending ISAs. The pre-process will be conducted on codes once the executable binary is loaded into memory, and the real-time control flow verification based on LEA can be done while ret and jmp instructions are executed. Performance evaluations on benchmarks show that the proposed method only introduces 0.61% run-time overhead and 0.63% memory overhead on average.

Categories and Subject Descriptors
B.8.1 Hardware [Performance and Reliability]: Reliability, Testing, and Fault Tolerance

General Terms
Security; Performance; Encryption

*Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC ’16, June 05-09, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4236-0/16/06... $15.00
DOI: http://dx.doi.org/10.1145/2897937.2898061

1. INTRODUCTION
In the past decade, one of the most prevalent attacks was code injection where attackers utilize the buffer overflow vulnerability to inject malicious codes on the stack and then overwrite the return address of a function with the address of the maliciously injected codes. Data execution prevention (DEP) [1] was proposed to prevent buffer overflow based attacks and has been widely used in modern operating systems. DEP marks memory as “executable” or “non-executable” in order to prevent attackers from directly executing malicious codes injected in the stack. However, attackers can redirect the control flow of programs by reusing small fragments of existing codes (called gadgets), thus eliminating the need of code injection and defeating DEP. This is known as code reuse attacks (CRAs) [2-7]. CRAs have been demonstrated on a broad range of architectures, such as x86, Atmel AVR, SPARC, ARM, Z80 and PowerPC [2, 4, 6].

Many defenses have been proposed to thwart CRAs recently. The most general solution is control-flow integrity (CFI) [7] approach, where the control flow graph (CFG) [7] of the program is generated during compilation and is enforced at run-time [11]. However, this has high memory and performance overheads that make it impractical in many applications. Therefore, recent research aim at balancing the tradeoff between security and such overheads by revising the software or modifying processor’s instruction set architecture (ISA). This transforms the performance overhead to the extra cost of software development and hardware adoption of the new ISA instructions.

In this study, we propose a new hardware-based CFI that combines silicon physical unclonable functions (PUF) [13] and an XOR-based lightweight linear encryption architecture (LEA) to reduce the cost of defending CRAs.

In our proposed PUF-based LEA, instructions at the destination of indirect jmp instructions and the return addresses of function-call are encrypted before they are stored in the memory structure. They will be decrypted when the corresponding instructions are executed. The encryption and decryption are performed at run-time by
an on-chip hardware unit with the XOR operation. The encryption and decryption keys are generated by silicon PUF. This approach has the following features:

1) The PUF-based LEA can protect programs from both types of CRAs: return-oriented programming (ROP) attacks and jump-oriented programming (JOP) attacks.

2) The PUF-based LEA does not need to revise the software or add any new instructions to the existing processor’s ISA.

3) Simulation results show that our architecture’s average run-time overhead is 0.61% and memory overhead is 0.63%, which is much lower than traditional CFI approaches.

4) The PUF keys are hardware-specific and can be used to bind programs to hardware. The key from a success attack to one system cannot be used to break another system.

The reminder of the paper is organized as follows. Section 2 introduces the basics on CRA with an example. Section 3 surveys related work on CRA defenses. Section 4 elaborates the proposed PUF-based linear encryption architecture (LEA). Section 5 reports the simulation results. Section 6 gives the analysis of the security and compatibility of the proposed method. Finally, we conclude in section 7.

2. CRA BASICS AND AN EXAMPLE

CRAs mainly include return-oriented programming (ROP) attacks [2, 4-7] and jump-oriented programming (JOP) attacks [3, 7]. ROP makes use of gadgets that end with the ret instruction to control programs. By overwriting the stack with the addresses of the gadget chain, attackers can change the execution flow of programs (see the example below). The gadget chain normally consists of short instruction sequences from different locations in the memory. Each sequence ends with a ret instruction to enable the chained execution of multiple instruction sequences [6]. The target address of a ret instruction is the address pushed into stack. Values in register, memory and stack can be changed by exploiting the overflow vulnerabilities, which remains prevalent.

Figure 1 illustrates an example of ROP attacks where the attacker’s goal is to make a system call “int 0x80” with parameter “data” in both eax and ebx. To achieve this goal, the attacker will first find two gadgets that write to eax and ebx such as the instructions “mov ecx, eax” and “mov ecx, ebx” (gadget2 and gadget3 as shown in Figure 1), respectively. Then the attacker will look for instruction such as “pop ecx” (gadget1) that can store “data” into ecx. Now the attacker can exploit the overflow vulnerability to store the addresses of the system call “int 0x80”, gadget3, gadget2, and the data in the stack in that order.

At run-time, after gadget1 is executed, CPU pops “data” from the stack into ecx. Then, the addresses in the stack will take the program flow to gadget2 and gadget3 to move the value “data” in ecx to eax and finally make the desired system call “int 0x80” with the correct “data”.

In JOP attacks, attackers can chain gadgets by the indirect jmp instructions. The gadgets are classified as control gadgets and function gadgets. Function gadgets are used to achieve attacker’s goal such as making a system call. Control gadgets are used to change the value stored in the registers to transfer the control flow to function gadgets. Control gadgets constitute a dispatcher to control the execution order of function gadgets. By deliberately organizing the executing order of the function gadgets, attackers can control the execution flow of the program and perform malicious actions.

3. EXISTING DEFENSE AGAINST CRA

Many defenses have been proposed to thwart CRAs, such as space layout randomization (ASLR) [2, 4], protecting stack [5], checking gadgets [2, 3, 6] and CFI [7, 9, 10].

ASLR [2, 4] randomizes the address of code and data regions when the program is loaded into memory in order to prevent attackers from guessing the entry address of gadgets. However, the code and data region is not fully randomized when ASLR is applied on modern operating systems. Besides, with the knowledge of some randomized codes, it is still possible to find enough gadgets in memory to perform CRAs. Protecting stack [5] was proposed to protect the stack of call/ret from being changed by adding a microprocessor. However, this method is very costly and complicated, and cannot defend against JOP attacks.

Checking gadgets [2, 3, 6] was proposed to monitor the frequency of executing gadgets to judge whether the program is under attack. This method can resist the JOP attack but it may incur a high false positive rate when a program consists of many small functions having little amount of instructions. CFI is the most general solution among current defenses. The key idea is to generate the control flow graph (CFG) for a program during compilation and enforce the control flow to follow the CFG at run-time. CFI include software-based CFI and hardware-based CFI as we will introduce in details next.

3.1 Software-based CFI

In the implementation of CFI [2], IDs and ID-checking instructions are inserted for each legal return/jump transitions in binaries and then checked while executing ret and jmp instructions. Any illegal changes of control flow will be theoretically checked and refused in run-time due to an ID-checking violation.

Figure 2 shows an example of CFI [2]. There are two kinds of control flow transitions in this example. The first one is the function pointer fptr pointing to function less than or greater than (in function sort), and the other is the function return of sort_1, sort_2, sort, less than, or greater than. In order to avoid any malicious modification on the target address of fptr due to ROP or JOP attacks, CFI adds the ID1 checking (check ID1) for verifying legal jumps from the source (function sort) to the destinations of fptr (functions less than and greater than). Any illegal jumps to other destinations will not pass the check ID1 because there are no ID1s inserted at those target addresses.
Meanwhile, CFI also adds the ID2 checking (check ID2) for verifying legal returns from the call functions to the callsite to avoid malicious modifications on the return addresses in the stack. Any changes to other return addresses will not pass check ID2 because there are no ID2s inserted at illegal addresses.

Theoretically, a unique ID can be inserted for each control flow transition. However, this would result in large performance degradation due to the ID creating, querying, comparing and storing. The performance overhead of CFI is about 21% [2]. In order to improve performance, several techniques [7, 9, 10] choose to loosen the control transitions and use least IDs. Compact control flow integrity and randomization (CCFIR) [7-9] redirects indirect jmp branches to a new springboard. By assigning aligned entry to direct and indirect branch targets, indirect jmp instructions will be checked and only allowed to transit control flow to the springboard entries. CCFIR use three IDs to restrict control flow. Two IDs are used to return to sensitive or non-sensitive functions, and one ID is used for call or indirect jmp instructions. This looser CFI allows control flow to jump to the address that does not present in program’s CFG, which makes it be possible for attackers to lunch an advanced ROP attack. Besides, CCFIR requires program allocating aligned springboard to ensure control flow integrity, which largely increases the space requirements.

As discussed above, current CFIs need to insert checking instructions or create accompanying data structures (e.g. stack) in executing programs, which may overwrite the registers or flags at run time and cause the programs behaving abnormally [10, 11]. In addition, software-based CFIs incur high performance and memory overheads [10]. Hence, the tradeoff between the security and the overhead is the key issue to be solved.

3.2 Hardware-Based CFI

Several hardware-based CFI methods were also proposed, such as Branch Regulation (BR) [6, 7, 9], hardware-assisted fine-grained CFI [10, 14].

BR adds hardware support for control flow checking, in which indirect jmp instructions are restricted to jump to its own function or the first instruction of other functions. BR also adds a shadow stack to record legal return addresses and check before each ret instruction. To improve efficiency, BR employs cache for accessing the shadow stack. However, branch regulation by design does not support stack unwinding and tail jumps [14].

Lucas et al. [10] proposed a hardware-assisted fine-grained CFI which adds new CPU instructions to ISAs. By assigning different labels to each function, it ensures an indirect call instruction must fit new CPU instructions. Ret instructions can only return to the most recent callsite since the label for call function can be activated at call time and inspected at return time. But labels are needed to insert into binary which needs to change the compiler and ISAs to support the recognition of labels. Besides, its run-time checking mechanism requires a label state memory [10] to store function labels, which increases the space overhead.

4. PUF-BASED LINEAR ENCRYPTION ARCHITECTURE

Instead of inserting IDs and ID checking instructions, we propose a lightweight linear encryption architecture (LEA) which encrypt and decrypt return addresses and some bytes of the first instruction of the target address during run-time to defend CRAs. The most significant advantage of the proposed LEA is that it will not extend the processor’s ISA. In addition, when generating the CFG, we only need the target addresses of indirect jmp instructions. What is more, LEA uses keys generated by silicon PUF. The keys will be unique for each system, making CRAs more expensive. Finally, LEA also is effective to prevent attackers from using intended instructions to achieve CRAs.

4.1 Architecture Overview

Figure 3 depicts the overview of the proposed linear encryption architecture (LEA), as shown in the dashed rectangle, and how it interacts with the CPU. LEA is an add-on hardware module to current CPU architecture. It has an encryption-decryption unit (EDU), two registers (KEY_CFI and LEN_CFI), and a silicon PUF module. The traditional architecture is connected with the EDU only. The two registers and PUF work with EDU internally. The encryption/decryption key is generated from the PUF module and stored in KEY_CFI. Only part of the key will be used for encryption/decryption. The length of the key used is determined by the requirements of both call-return and jump encryptions and stored in LEN_CFI. In this study, it is set less than the lengths of an address.

Figure 2: An Example of CFI [2].

Figure 3: LEA and data flows. The data flows are: 1) the encrypted instructions for jmp, 2) the decrypted instructions for jmp, 3) the return addresses to encrypt for call or the instructions to encrypt/decrypt for jmp 4) the decrypted addresses for ret 5) the encrypted addresses for call in stack.
and the shortest instruction of all the first instruction of target address.

The first task of EDU is to encrypt return addresses and some bytes of the first instruction at jmp destinations. When the program is loaded into memory, pieces of the first instruction at jmp destination are all encrypted. The return addresses are encrypted as soon as they are pushed into the stack. The second task of EDU is the corresponding decryption. For ret instructions, EDU decrypts the return addresses popped from stack and transfers them to the Instruction Pointer (IP) to follow up properly. For indirect jmp instructions, EDU decrypts the specified bytes (determined by LEN_CFI register) of the first instruction at destination addresses of jmp, and then transfers them to Instruction Register (IR) to follow up. Figure 3 illustrates five detailed ways that EDU will communicate with CPU and memory for encryption/decryption.

The key stored in KEY_CFI is generated by a PUF module, which can generate a random, hardware-unique and unclonable key. Any key updating only can be triggered by the CPU according to predefined schemes.

For traditional CFIs, the inserted ID-checking instructions may influence caching, branch predictions and even correct behaviors due to the change of runtime registers or flags. Our proposed architecture will not have these problems except for the increased execution time of the related instructions, i.e. call, ret, jmp.

### 4.2 Linear Encryption and Its Overhead

We propose to use a linear encryption method in the EDU to reduce its performance overhead and impact to CPU architecture. Let E be the data to be encrypted (i.e., the return address or instructions at jmp destinations) and K be the key stored in KEY_CFI. We simply encrypt the data by XOR-ing E and key K:

$$A = E \oplus K$$  \hspace{1cm} (1)

When CPU executes a call instruction, the return address will be transmitted to the EDU for encryption and the encrypted address will be pushed into stack. When the CPU executes a ret instruction, it pops the encrypted address from the stack and transmit it to EDU for decryption. Then the program returns to the decrypted address to perform. When the CPU executes an indirect jmp instruction, it transmits the encrypted instruction with the length of LEN_CFI at the destination address of the jmp to EDU to decrypt and then goes on execute instructions.

Given a program with n call instructions and m indirect jmp instructions, we assume that 1) the average instruction execution time is \( t \) and 2) the encryption and decryption time of EDU are also \( t \) because they are all linear operations based on XOR which should run faster than average instructions. Therefore, we can estimate the linear encryption architecture’s runtime overhead as

$$\text{overhead} = 2nt + 4mt = (2n + 4m)t$$  \hspace{1cm} (2)

where \( 2nt \) is the total run time of encryption and decryption operation for each call-ret pair, and \( 4mt \) is the run time of the two inserted jmp instructions (\( 2mt \)) and the encryption and decryption operations (\( 2mt \)) for the jmp instructions (encryption is in the preprocess while loading into memory). Hence, the runtime overhead of the proposed method is \( O(n + 2m) \), which means a linear overhead. However, compared with the traditional CFI, our proposed method has a big advantage in the runtime overhead. On the other hand, there are two instructions (mov and jmp) added at each jmp destination and there are no other memory needed for decryption and encryption. The memory overhead is thus \( O(m) \).

![Figure 4: An Example of call-return instructions using LEA.](image)

### 4.3 Call-Return Instructions

Attackers often overflow the stack to modify the return address and then chain the gadgets to launch a ROP attack. In order to resist such unauthorized modifications on return addresses, we propose to encrypt the addresses using a PUF generated key. In our proposed method, when a call instruction is executed, the CPU transmits the return address to EDU for encrypting before storing it into stack. EDU uses the key in the KEY_CFI register generated by a PUF module, and the length of the key is determined by the LEN_CFI register. When the corresponding ret instruction is executed, CPU pops the return address from stack and transmits it to EDU to decrypt. If attackers modify the return address without encrypting with the correct key, the execution will go wrong. Figure 4 shows an example for call-ret instructions using our proposed encryption method.

Normally, after a call instruction is executed, CPU pushes the return address of 004095F6h into the stack and returns to execution after a corresponding ret instruction is executed. In our method, the EDU encrypts 004095F6h with the key and pushes the encrypted address (0E5423FEh) into the stack. If attackers modify the encrypted address in the stack, the default decryption operation for the address will be forced to perform, which will make the control flow abnormal and finally may raise a system error.

### 4.4 Indirect Jump Instruction

In addition to ret instructions, attackers can use indirect jmp instructions to hijack the control flow. To defend against this kind of attacks, we propose to encrypt some bytes of the first instruction at the target address of jmp as soon as the program is loaded into memory. The key is also from the register LEN_CFI and the length of key is determined by the LEN_CFI register. While the CPU executes a jmp instruction, the first instruction with the length determined by LEN_CFI at the target address will be decrypted using the same key from KEY_CFI. EDU then puts the decrypted result to the IR. If attackers change the target address to elsewhere, the instructions there would be forced to be decrypted, if an incorrect decrypted instruction is executed, a system error may occur. Figure 5 shows an example of the jmp instructions using LEA.

In this example, we assume the value of LEN_CFI is 3 and the target address of jmp ecx instruction is 004095F6h. CPU encrypts the first 3 bytes of the instruction at the des-
program can be executed correctly. Finally, we run these
instructions according to the CFG in order to guarantee the
inserted before each intended target of indirect
\texttt{jmp} instruction/decryption operator. An indirect
\texttt{jmp} instruction address when the program is loaded into memory.
Once indirect \texttt{jmp} instructions jump to any other unen-
crypted instructions, an unintended behavior or a system
error may occur due to the operation of default decrypt-
ment. Meanwhile, the LEA will encrypt all candidate \texttt{jmp}
destinations and triggers \texttt{jmp} decryption while executing
\texttt{jmp} instructions, but some instructions at candidate desti-
nation addresses of \texttt{jmp} may also be executed sequentially
(e.g. do-while or if-statement alone) without being the des-
tination of the \texttt{jmp}. In this situation, an exception/error
may occur. Therefore, we introduce a redundant \texttt{jmp} in-
struction before those instructions at candidate \texttt{jmp} desti-
nation addresses to judge the sequent execution of those
instructions, e.g. the \texttt{mov edx 004095F6h} and \texttt{jmp [edx]}
in this example.

4.5 Silicon PUF for Key Generation and Man-
gagement

The use of silicon PUF in LEA has several advantages.
First, it is secure. PUF is a more secure storage of secret
because it is hard for the attacker to steal or forge PUF
data. Second, it is lightweight. Given the small size of the
key, using PUF to generate key will require less hardware
than other approaches such as a random number gener-
ator. Third, PUF is based on fabrication variation, which
is believed to be unique from chip to chip. Therefore, even
when attackers break the key on one system, they cannot
apply the same key for other systems. What is more, the
PUF module in LEA is only used to generate the key when
the program is starting so that it has ignorable influence
in the performance.

5. EXPERIMENTAL RESULTS

The linear encryption architecture and the performance
overhead are evaluated on nine different types of programs.
The experimentation was conducted on a 32-bit Intel X86
Core I5 CPU with 4GB memories with an Ubuntu 12.04
OS.

To evaluate the performance overhead, the profiling of
the program is done in order to obtain the CFG at the
compile time. Then a XOR instruction is inserted before
every \texttt{call/ret} and indirect \texttt{jmp} instruction for the encryp-
tion/decryption operator. An indirect \texttt{jmp} instruction
is inserted before each intended target of indirect \texttt{jmp}
instruction according to the CFG in order to guarantee
the program can be executed correctly. Finally, we run these
programs and summarize the performance overhead. Ta-
ble 1 shows the time and space overhead of our proposed
method on nine different programs. The average memory
overhead is 0.63%. The average run time overhead is about
0.61%. 0.86% loading time overhead is incurred in load-
programs due to the preprocess of \texttt{jmp} instructions.
The reason of extremely low time and memory overheads
is that LEA has linear processing complexity (proportion-
tal to \texttt{#jmp} and \texttt{#ret}) and limited memory requirements
(proportional to \texttt{#jmp}). LEA does not need to insert IDs
and check IDs, which also greatly reduces CFI’s time and
space complexity.

6. SECURITY AND COMPATIBILITY ANAL-
YSIS

The potential security issues for our proposed LEA are
analyzed as follows.

6.1 Security Analysis

6.1.1 The key leakage issue

We employed XOR logic operations to encrypt and de-
crypt the return addresses in LEA, which may be attacked
by chosen plaintext attacks (CPA) since the attackers may
get the linearly encrypted results through memory leakage
or debugging. This does not damage the defense to JOP
attacks of CRA because the attackers cannot modify the
executable binaries at run time when they do not have sys-
tem control (basic assumption). Additionally, this can be
mitigated by randomly choosing PUF keys at program’s
start-up, or dynamically updating the PUF keys at regular
intervals. Dynamically updating the key would make the
attack be harder since attackers must focus on each chip
to launch an attack due to the uniqueness of PUF key for
each chip, and even though the attackers can get the key
luckily, but the key may be updated at regular intervals.
In other words, a successful attack on a chip cannot be
easily applied to another one due to the dynamic key.

6.1.2 False positive/negative rate

The false negative rate and false positive rate are two im-
portant metrics to evaluate the security and reliability of a
computer system. The same with all other CFI methods,
without the complete CFG, our proposed method might
incur false positive at the runtime (a false positive would
mean a normal control flow is detected as a malicious con-
trol flow). As a future work, we can use advance program
analysis to increase the coverage of CFG. In addition, the
attackers can find possible gadgets among legal \texttt{jmp} des-
tinations since they do not distinguish whether the jump
sources is legal in decryption. Hence, our method may
incur false negatives (a false negative would mean a malici-
ous control flow that is not detected). We evaluate the
false negative rate on benchmarks. We can see from Ta-
ble 3 that the available gadgets are reduced about 91.92%
on average by using LEA for \texttt{jmp} instructions. The prob-
ability of achieving a successful jump using those survived
gadgets is decreased largely. Table 2 gives the number of
encryption and decryption operations of \texttt{call, ret} and in-
tend target of \texttt{jmp} instructions for different programs ob-
tained from simulation. The simulation results show that
the false negative rate is less than 1%.

6.2 Compatibility Analysis

Most of traditional CFIs have the compatibility issues.
The hardened modules and un-hardened modules cannot
interoperate, which prevents incremental deployment that
is often needed in real systems [8]. A further challenge is

Figure 5: An Example of indirect \texttt{jmp} instructions using LEA.
7. CONCLUSION AND FUTURE WORK

Traditional control flow integrity (CFI) methods incur high memory and performance overheads or need to modify ISAs against code reuse attacks (CRA). In this study, we propose a hardware-based CFI that employs PUF-based linear encryption architecture to resist CRA with extremely low hardware and run time overheads. It can protect ret and indirect jmp instructions from ROP and JOP without any additional software manipulations and extending ISAs. The pre-process on code is conducted as soon as the executable binary is loaded into memory, and the real-time control flow verification based on LEA is done while restricted jumps are, but any overlapping points-to-sets must be unified to use the same ID [8].

8. ACKNOWLEDGEMENTS

This work is supported in part by the Tsinghua University Initiative Scientific Research Program, the AFOSR MURI under award number FA9550-14-1-0351, the National Science Foundation for Distinguished Young Scholars of China under Grant No. 61225012 and the State Key Development Program for Basic Research of China under number 2012CB315801.

9. REFERENCES

Table 1: Time and space overhead of our proposed method on nine different category programs

<table>
<thead>
<tr>
<th>Category</th>
<th>Program</th>
<th>Size(B)</th>
<th>Memory overhead(%)</th>
<th>Loading time(s)</th>
<th>Run time overhead(%)</th>
<th>Result(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>Vol</td>
<td>10894</td>
<td>1</td>
<td>100.00</td>
<td>27</td>
<td>14</td>
</tr>
<tr>
<td>Game</td>
<td>Mines</td>
<td>720062</td>
<td>2</td>
<td>99.04</td>
<td>1218</td>
<td>305</td>
</tr>
<tr>
<td>Plotting</td>
<td>Orange</td>
<td>1270085</td>
<td>4</td>
<td>92.54</td>
<td>1331</td>
<td>400</td>
</tr>
<tr>
<td>Picture</td>
<td>Erg</td>
<td>2337126</td>
<td>47</td>
<td>94.81</td>
<td>3391</td>
<td>1212</td>
</tr>
<tr>
<td>IO</td>
<td>Hardinfo</td>
<td>2555039</td>
<td>58</td>
<td>95.06</td>
<td>3945</td>
<td>2529</td>
</tr>
<tr>
<td>Text</td>
<td>Gedit</td>
<td>2751830</td>
<td>126</td>
<td>87.84</td>
<td>4607</td>
<td>1271</td>
</tr>
<tr>
<td>Mail</td>
<td>Musk</td>
<td>2395761</td>
<td>464</td>
<td>87.91</td>
<td>7115</td>
<td>315</td>
</tr>
<tr>
<td>Chat</td>
<td>Xchat</td>
<td>3656663</td>
<td>203</td>
<td>90.67</td>
<td>5492</td>
<td>831</td>
</tr>
<tr>
<td>Player</td>
<td>SPlayer</td>
<td>7551334</td>
<td>265</td>
<td>94.77</td>
<td>19728</td>
<td>560</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>22195354</td>
<td>1658</td>
<td>91.92</td>
<td>46764</td>
<td>7165</td>
</tr>
</tbody>
</table>

Table 2: Gadgets reduction and false negative with LEA

<table>
<thead>
<tr>
<th>Category</th>
<th>Program</th>
<th>Size(B)</th>
<th>Gadgets</th>
<th>Reduction gadgets</th>
<th>Instruction gadget</th>
<th>Result(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>Vol</td>
<td>10894</td>
<td>1</td>
<td>100.00</td>
<td>27</td>
<td>14</td>
</tr>
<tr>
<td>Game</td>
<td>Mines</td>
<td>720062</td>
<td>2</td>
<td>99.04</td>
<td>1218</td>
<td>305</td>
</tr>
<tr>
<td>Plotting</td>
<td>Orange</td>
<td>1270085</td>
<td>4</td>
<td>92.54</td>
<td>1331</td>
<td>400</td>
</tr>
<tr>
<td>Picture</td>
<td>Erg</td>
<td>2337126</td>
<td>47</td>
<td>94.81</td>
<td>3391</td>
<td>1212</td>
</tr>
<tr>
<td>IO</td>
<td>Hardinfo</td>
<td>2555039</td>
<td>58</td>
<td>95.06</td>
<td>3945</td>
<td>2529</td>
</tr>
<tr>
<td>Text</td>
<td>Gedit</td>
<td>2751830</td>
<td>126</td>
<td>87.84</td>
<td>4607</td>
<td>1271</td>
</tr>
<tr>
<td>Mail</td>
<td>Musk</td>
<td>2395761</td>
<td>464</td>
<td>87.91</td>
<td>7115</td>
<td>315</td>
</tr>
<tr>
<td>Chat</td>
<td>Xchat</td>
<td>3656663</td>
<td>203</td>
<td>90.67</td>
<td>5492</td>
<td>831</td>
</tr>
<tr>
<td>Player</td>
<td>SPlayer</td>
<td>7551334</td>
<td>265</td>
<td>94.77</td>
<td>19728</td>
<td>560</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td>22195354</td>
<td>1658</td>
<td>91.92</td>
<td>46764</td>
<td>7165</td>
</tr>
</tbody>
</table>

diversity of IDs. The more IDs the code uses, the more restricted jumps are, but any overlapping points-to-sets must be unified to use the same ID [8]. Sharing of jump targets such as library functions can lead to many sites having only one ID [8].

In this study, LEA adds linear encryption and decryption operations to specific instructions including call, ret and jmp, which can be implemented by a XOR combinational logic circuit. When implementing the LEA on current general architectures, we can model the encryption and decryption process as new operations attached to the execution of those related instructions, e.g. an extra cycle in pipeline. This method would introduce a non-parallel cycle in pipelines and incur negligible performance overhead.