Abacus: A Tool for Precise Side-channel Analysis

Qinkun Bao*, Zihao Wang*, James R. Larus†, and Dinghao Wu*
*The Pennsylvania State University
†EPFL

***Abstract*—Side-channel vulnerabilities can leak sensitive information unconsciously. In this paper, we introduce the usage of Abacus. Abacus is a tool that can analyze secret-dependent control-flow and secret-dependent data-access leakages in binary programs. Unlike previous tools that can only identify leakages, it can also estimate the amount of leaked information for each leakage site. Severe vulnerabilities usually leak more information, allowing developers to triage the patching effort for side-channel vulnerabilities. This paper is to help users make use of Abacus and reproduce our previous results. Abacus is available at https://github.com/s3team/Abacus.**

I. Introduction

Abacus [1] is an address-based side-channel vulnerability analysis tool. Different from previous tools [2]–[4], it can also give a precise estimation of the amount of the leaked information for each leakage site. Abacus is open source under the MIT License.

Abacus takes a binary executable as the input. It uses dynamic binary instrumentation tools to collect execution traces. After that, Abacus analyzes those traces and produces the vulnerability report. While Abacus can work on the stripped binary executable, Abacus can also read the symbol and debugging information to give a more fine-grained (e.g., line numbers) report. Table I shows an example of the report. If you are interested in how Abacus works, please refer to the technical paper [1].

Table I: A sample leakage report

File	Line No.	Function	# Leaked Bits	Type
set_key.c	350	DES_set_key_unchecked	5.8	DA
set_key.c	350	DES_set_key_unchecked	6.6	DA
set_key.c	350	DES_set_key_unchecked	7.5	DA
set_key.c	350	DES_set_key_unchecked	6.4	DA
set_key.c	355	DES_set_key_unchecked	1.9	DA
set_key.c	355	DES_set_key_unchecked	3.1	DA

II. Requirements

We have tested Abacus on both macOS and Ubuntu. You can refer to the continuous integration scripts to build Abacus on your operating system. However, we strongly recommend you to build Abacus inside a container to avoid any dependency problems. To simplify the illustration, we only include the instructions of installations within the docker in this paper.

- Supported OS: Ubuntu 18.04
- Memory: 32 GB (If you want to run experiments concurrently, update the size of RAM accordingly. Otherwise the program may be terminated by the system.)

III. Installation

Abacus can be built within a docker, simply run the following command:

```
$ git clone https://github.com/s3team/Abacus.git
$ cd Abacus
$ ./docker.sh
```

The "docker.sh" script creates a docker image automatically and enters the container that includes all dependencies. After that, run the following command to build Abacus:

```
$ ./build.sh
```

IV. Run the hello world example

In this section, we walk you through the steps to test a simple function with Abacus.

```
1  #include <stdio.h>
2
3  void is_odd(uint16_t secret) {
4    int res = secret % 2;
5    if (res) {
6      printf("Odd Number\n");
7    } else {
8      printf("Even Number\n");
9    }
10 }
```

Figure 1: A simple example

As shown in Figure 1, the function takes a 32-bit integer as the secret input and checks the last digit of the integer. An attacker can know the last bit of the input integer by observing which branch is actually executed. So in the above function, we think the code has one secret-dependent control-flow vulnerability, and it can leak one bit of the secret information.

A. Mark secret data as symbolic

In order to test this function with Abacus, we need to mark the variable that representing the secret data as a symbolic variable. We use the function abacus_make_symbolic. The function takes three arguments: the type of the symbol, the address of the secret, and the length of the secret input. In the below example, the secret input is the variable secret, and its length is two bytes. We add the main function in Figure 2 and compile the source code into an executable.

```
12  int main() {
13    uint16_t secret = 6;
14    char *type = "1";
15    abacus_make_symbolic(type, &secret, 2);
16    is_odd(secret);
17    return 0;
18  }
```

Figure 2: A simple function that marks a variable secret as symbolic

B. Build the example

Abacus analyzes vulnerabilities on the binary executable. Here we build it into a 32-bit ELF executable. Note that while Abacus can work on stripped binaries without the source code, we use debug information to get a more detailed result (e.g., the line number in the source code) in this example.

```
$ cd examples
$ gcc -m32 -g example1.c
```

C. Collect the trace

We use the pin tool to collect the execution trace. The tool can automatically collect the trace and other necessary runtime information.

```
$ cd /abacus/Pintools
$ make PIN_ROOT=/abacus/Intel-Pin-Archive/ TARGET=ia32
$ cd /abacus
$ /abacus/Intel-Pin-Archive/pin \
  -t Pintools/obj-ia32/MyPinToolLinux.so \
  -- ./examples/a.out
```

You will get two files Function.txt and Inst_data.txt. Inst_data.txt is the mandatory input of Abacus. Function.txt is optional.

D. Quantify the leakage

To analyze the execution trace and generate the report, run the below command:

```
$ ./build/App/QIF/QIF ./Inst_data.txt -f Function.txt \
  -d ./examples/a.out -o result.txt
```

You should get the following output:

```
Start Computing Constraints
Total Constraints: 1
Control Transfer: 1
Data Access: 0
Information Leak for each address:
Address: 5664259b Leaked:1.0 bits Type: CF
Source code: example1.c line number: 3
Function Name:  is_odd Module Name:  a.out Offset: 30
...
```

As expected, it shows that the function is_odd has one secret-dependent control-flow vulnerability at line 5. Also, Abacus shows the vulnerability leaks 1 bit information.

V. Analyze Cryptography Function

We have applied Abacus on the following libraries:

- OpenSSL: 0.9.7, 1.0.2f, 1.0.2k, 1.1.0f, 1.1.1, 1.1.1g
- MbedTLS: 2.5, 2.15
- Libgcrypt: 1.8.5
- Monocyper: 3.0

The results can be reproduced by running the simple command after you build Abacus successfully inside the container. We have prepared scripts to analyze each cryptography algorithm automatically. For example, if you want to test AES in mbedTLS 2.5, you can simply run the following command.

```
$ cd /abacus/script/AES_MBEDTLS_2.5
$ ./start.sh
```

VI. Command-line Options

Abacus takes the trace file as the input (Inst_data.txt). Besides the trace file, Abacus has the following command-line options:

-d <executable file>
Read an elf *executable file*. Abacus can parse the debug information inside the file. With the optional input, Abacus is able to output which line in the original source code actually leaks the sensitive information.

-f <function file>
Read a function file that was generated by the Pin tool. The command is optional. With the optional input, Abacus is able to output which function leaks the information and call sites from the sensitive buffer to the leaked site.

-n <Monte Carlo times>
Set the times of Monte Carlo Sampling when Abacus estimates the amount of leakage information. If you do not specify the option, Abacus can automatically terminate the sampling when the tool has 95% confidence that the error of estimated leaked information is less than 1 bit. Please refer to the paper if you want to learn more about the details.

-a <the address of the secret buffer>
-s <the size of the buffer>
The two input options must be used together. In the previous example, we use abacus_make_symbolic to mark the secret buffer. For a raw binary, we use the above two options to tell Abacus which buffer is the secret and its length.

Acknowledgment

We thank anonymous reviewers for their valuable feedback. The work was supported in part by the National Science Foundation (NSF) under grant CNS-1652790, and the Office of Naval Research (ONR) under grants N00014-16-1-2912, N00014-16-1-2265, and N00014-17-1-2894.

References

[1] Q. Bao, Z. Wang, X. Li, J. R. Larus, and D. Wu, "Abacus: Precise side-channel analysis," in *ICSE 2021*.

[2] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu, "CacheD: Identifying cache-based timing channels in production software," in *USENIX Security 17*.

[3] S. Wang, Y. Bao, X. Liu, P. Wang, D. Zhang, and D. Wu, "Identifying cache-based side channels through secret-augmented abstract interpretation," in *USENIX Security 19*.

[4] G. Doychev, D. Feld, B. Kopf, L. Mauborgne, and J. Reineke, "CacheAudit: A tool for the static analysis of cache side channels," in *USENIX Security 13*.