Automated Testing against Timing Attacks

Justin Dove

Victor Vasiliev

ABSTRACT

This paper provides an overview of a system which allows to integrate detection of potential timing attack with a regular test suite. The authors examine previous approaches to the problem and discuss their usefulness and fitness for the purposes of automated testing. A developer-friendly system for detecting timing issues using Valgrind is presented, and integration with Google Test is discussed. An example of how such test can detect vulnerabilities is provided, and the limitations of the system are outlined.

1. INTRODUCTION

Side-channel attacks are attacks which compromise security of communication by analyzing the information that is not explicitly encoded in the channel. Such attacks rely on various sources of information: consumption of time, power, emission of electromagnetic radiation, and many others. The timing attacks are the most common type of attacks encountered at the software level, and thus we focus on them.

Timing attacks exploit the difference in time that it takes for a program to perform computation on private data. This kind of attacks dates back to the mainframe days. In TENEX operating system, the attacker could run a password check on a carefully positioned fragment in memory and measure the time it takes to check the password. If the password string was positioned across a page boundary, the password check would take longer. That was sufficient for the attackers to determine how many characters in supplied password was correct, and thus bruteforce the password in linear time [10].

The performance of modern computers has dramatically increased since TENEX days, making straightforward attacks less feasible. However, the exposure surface is still there. Previous research in this area has indicated that the timing attacks are practical [4], and that there are numerous features of modern CPU designs that make those attacks hard to mitigate [3]. We will demonstrate our own measurement showing to what extent it is indeed possible to perform a timing attack on a modern Haswell Intel CPU, and suggest a method of how to diagnose those issues.

```
static std::string target =
  "Time is an illusion, lunchtime doubly so.";

bool BlackBoxCheck(const std::string &attempt) {
  if (target.size() != attempt.size()) {
    return false;
  } else {
    for (size_t i = 0; i < target.size(); i++) {
      if (target[i] != attempt[i]) {
        return false;
      }
    }

    return true;
  }
}
```

Figure 1: An example of a function insecure against timing attacks.

2. FEASIBILITY OF TIMING ATTACKS

We have attempted to perform a timing attack on a simple C++ function which checks the equality of a string with a reference one in an unsafe manner: it returns as soon as it discovers first mismatch (figure 1). The attack is as follows:

1. Use timing oracle to find the correct string length, since the function returns immediately in case of length mismatch.
2. Guess first letter three times in a row. If they yield a consistent result, output it, otherwise continue.
3. Continue guessing until all the letters are guessed.

Note that this algorithm is not guaranteed to terminate. In fact, it will not, if the timing oracle data is not distinguishable from noise. For the implementation of the timing oracle, we use C++11's built-in high resolution clock (which on Linux is aliased to the system clock).

We have ran the algorithm on a i7-4770 desktop running Debian, and got consistent results for an unoptimized build

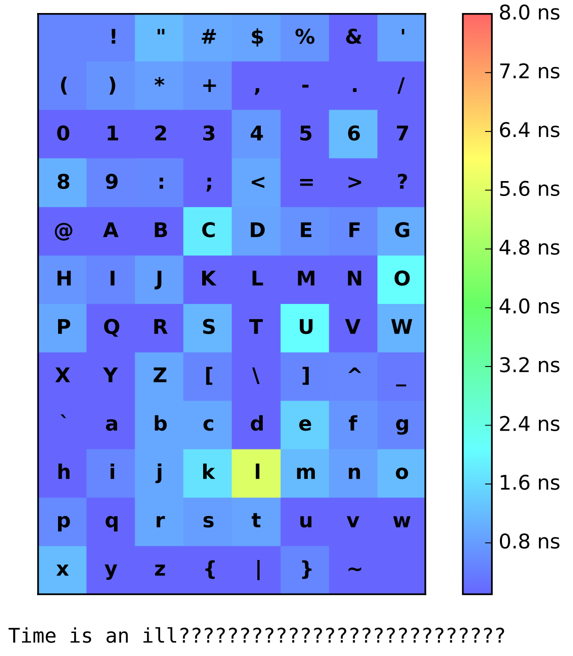

Figure 2: Deviation from the median time for each letter at $i = 14$

of the program and the communication with the oracle handled via function call (figure 2). We have discovered that it becomes harder to perform the attack if ran on laptop, due to the aggressive power saving mechanisms which alter the clock frequency dynamically. The test did not work reliably over local Unix domain sockets, nor did it work with an optimized build. By inserting a sleep in the middle of the function, however, we've discovered that it still can work, except the difference in the amount of instructions executed should be an order of magnitude higher (which means we would need at least 100 cycles to make difference). We believe that using `RDTSC` instruction would make a local timing attack more powerful, as it has higher resolution than the system clock.

Overall, this suggests that timing attacks are still viable, especially in case of local or co-hosted attackers. An effective remote timing attack has been performed at least as recent as 2013 [2]. In addition, there exists amplification techniques which we did not explore [12].

3. TESTING

Our conclusion so far is that timing attacks are, in fact, a threat; this does not necessarily immediately imply that defending against them is a worthwhile investment. Such defense may be complex, and in many cases it comes at performance penalty, severe enough to render many straightforward fixes impractical. Even when solving the problem is feasible, it comes at a cost of increasing code complexity and higher maintenance burden. While the body of literature on those attacks is considerable, there has been very few high profile cases of security breaches achieved through them. In fact, CVEs for timing attacks normally are marked as having a fairly low level of severity, due to high level of technical skills required to exploit them successfully.

We believe that defending against all attacks known is the only real way to achieve actual security of any system. Timing attacks, due to their nature, can easily go undetected, so it is entirely possible that advanced threat actors have beeen using them for quite a while without anyone noticing. Hence, we believe that it is important to provide software developers who engineer cryptographic solution with a set of practical tools for defending software against those attacks.

3.1 Choice of mechanism

There are two aspects of defending against timing attacks. One of them is devising techniques which make code constant time; those vary from bit-level hacks to complete replacement of cryptographic algorithms, and are out of the scope of this paper. Another is determining whether a code has a timing vulnerability. There are two primary approaches one can take: try attacking the code directly, or analyze the program in order to detect operations that are insecure from the timing standpoint. The first approach has an advantage of being able to tell whether an attack is *actually* feasible. Unfortunately, automating such tests is hard, since in many cases those require carefully chosen input values to be noticeable. While they can tell you whether you have a timing discrepancy for different inputs, they will not tell you where it comes from unless you carefully profile the code in question. This makes debugging complicated, and in general those tests are difficult to implement in practice.

An alternative would be to perform analysis of the code for timing attacks by tracking the values and reporting an error whenever data is possibly leaked through the timing side-channel. A classic form of *static analysis* would be formal verification against timing attack. The problem here is that in its current forms formal verification is not practically applicable in the production software, and is cumbersome, especially in terms of actually modelling side-channels. An alternative approach to static analysis would be to use a type system of a programming language to enforce constant-time constraints. Such methods were already proposed, both as a domain-specific language [11], and as an extension to C++ programming language [8]. The latter proposal shows some promise, although according to our understanding the level of effort required to even fully state, yet alone implement all the constraints within the framework of already highly complex language, is yet to be fully seen.

Instead, we chose the paradigm of *dynamic analysis*. Our approach is based on the idea of adapting Valgrind's uninitialized value tracker for detecting behavior based on private data, originally proposed by Adam Langley in [6]. Valgrind marks uninitialized regions of memory as "poisoned", and poisons all the bits that depend on the already poisoned ones. Then it reports errors whenever a poisoned value is branched upon, passed to a system call or dereferenced as an address in memory [9]. This encompasses most of the cases which leak timing information.

3.2 Testing tool

Our implementation follows the original `ctgrind` tool, but it does not require any modifications to Valgrind and is integrated with the test suite.

We have integrated Valgrind with Google Test framework in order to produce the timing tests as one depicted on figure 3. We provide a macro to define a timing test, and a pair of function `TimingPrivateMark` and `TimingPrivateUnmark`, which allow to mark the data as private or public, as desired.

Here, `equality_good` is a timing-safe string comparison which uses XOR on the bytes in each string; `equality_bad`, on other hand, uses naive comparison, as described in figure 1.

When ran, the test for `CompareTiming.Good` will pass, while `CompareTiming.Bad` will fail with the error showing in which file and on which line exactly the violation of the rules has happened (figure 4).

3.3 Testing on real implementations

We tested out our approach on some cryptographic code. libseal[1], an open source cryptographic library developed by one of the authors, implements multiple cryptographic algorithms and already uses gtest. We added some timing tests into it, and were able to confirm that the SHA-1 implementation and Intel AES-NI did not have any timing attacks detected, while plain-C reference implementation of AES was clearly vulnerable to them.

4. LIMITATIONS

Despite having significant advantage in terms of ease of use and rigor of validation, our approach naturally has some inherent disadvantages.

4.1 Accuracy of the model

Any form of the timing attack vector detection done through analysis of the code relies on the model of which operations would cause timing being leaked on different input data, and which would not. We stick to the model used in Valgrind for uninitialized values, which is a very good fit for our purposes. However, there exists an important omission: instructions where the execution time depends on the input.

For Intel CPUs, the list is documented in Intel Optimization Manual [1], and is well-known to contain, out of non-floating point operations, integer multiplication and division. Such gap has few discrete timings and in practice can be fixed by multiplying the value by a large number, and then dividing that number off [7].

We have prototyped a patch against Valgrind which would trigger error on those (outlined in Appendix A). That does not, however, actually address the issue fully. The first issue is that the same list has to be aware about all problematic instructions across all supported platforms. The second issue is that the CPU manufacturers do not actually provide any guarantees that the list of unsafe instructions will not change.

In addition, there might be some cases where Valgrind model might be too restrictive for us to use. For example, the documentation suggests that parameters determining vector permutations can cause errors, while in practice those might be used to implemented timing-safe cryptographic primitives [5].

[1]`https://github.com/vasilvv/libseal`

```
// An example of a good check
bool equality_good(const std::string &a,
                   const std::string &b) {
  if (a.size() != b.size()) {
    return false;
  }

  uint8_t result = 0;
  for (size_t i = 0; i < a.size(); i++) {
    result |= (a[i] ^ b[i]);
  }

  return !result;
}

// equality_bad works the same way
// as BlackBoxCheck

TEST(CompareValidity, Basic) {
  ASSERT_TRUE(equality_bad("test", "test"));
  ASSERT_TRUE(equality_good("test", "test"));

  ASSERT_FALSE(equality_bad("text", "test"));
  ASSERT_FALSE(equality_good("text", "test"));

  ASSERT_FALSE(equality_bad("test+", "test"));
  ASSERT_FALSE(equality_good("test+", "test"));
}

TIMING_TEST(CompareTiming, Good) {
  std::string input1 = "test";
  std::string input2 = "text";

  TimingPrivateMark(input1);
  TimingPrivateMark(input2);

  equality_good(input1, input2);
}

TIMING_TEST(CompareTiming, Bad) {
  std::string input1 = "test";
  std::string input2 = "text";

  TimingPrivateMark(input1);
  TimingPrivateMark(input2);

  equality_bad(input1, input2);
}
```

Figure 3: An example of timing attack test. Observe how it naturally fits within the same idioms that the regular tests use.

```
[ RUN      ] CompareTiming.Bad
/home/vvv/final-project/testing/tests.cc:30: Failure
Death test: CompareTiming_Bad_fork()
    Result: died but not with expected exit code:
            Exited with exit status 57
Actual msg:
[  DEATH   ] 50
[  DEATH   ] ==10865== Memcheck, a memory error detector
[  DEATH   ] ==10865== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
[  DEATH   ] ==10865== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
[  DEATH   ] ==10865== Command: /home/vvv/final-project/testing/build/timing_tests --gtest_filter=CompareTiming.Bad
[  DEATH   ] ==10865==
[  DEATH   ] ==10865== Conditional jump or move depends on uninitialised value(s)
[  DEATH   ] ==10865==    at 0x4143B3: equality_bad(std::string const&, std::string const&) (compare.cc:9)
[  DEATH   ] ==10865==    by 0x4147C5: CompareTiming_Bad_core() (tests.cc:37)
[  DEATH   ] ==10865==    by 0x416CD3: CompareTiming_Bad_Test::TestBody() (tests.cc:30)
[...]
```

Figure 4: Error shown when a timing issue is discovered.

4.2 False positives

Another issue with using this tool is that you have to be careful to explicitly state when your private data becomes public. That sounds obvious, but there are plenty of subtle cases. A common example would be RSA blinding, or the integer multiplication trick above: the data is still private in its nature, but for the purposes of the side channel attacks, it is secure. In those cases, you would have to manually mark the data as public before performing an insecure operation, and then manually mark it as private again. The latter is dangerous, as in many cases the data, even in "blinded" form, is still private, and unblinding it may not cause it to automatically be marked as private.

5. CONCLUSIONS

This paper presented an approach of testing against presence of timing attack which is suitable for use in production cryptographic software. We believe that should it be adopted, it can make a profound impact on the amount of timing attacks in widely developed cryptographic software.

We are currently looking into implementing a full-scale test suite for one of widely used cryptographic libraries, with a hope of submitting it upstream.

6. ACKNOWLEDGMENTS

We thank Madars Virza for his advice and feedback on this paper. We are also grateful to Nickolai Zeldovich and David Lazar for their feedback on the original Decor idea, which helped us to choose the approach for designing our test mechanism. We thank Alex Chernyakhovsky for helpful discussion on the original Decor design draft and the practicalities of implementing it. We are grateful to David Benjamin for discussion on many real instances of timing attacks, which was a huge inspiration behind this project.

7. REFERENCES

[1] Intel 64 and IA-32 architectures optimization reference manual. `https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`, 2014.

[2] N. AlFardan and K. Paterson. Lucky 13: Breaking the TLS and DTLS record protocols. In *IEEE Symposium on Security and Privacy*, 2013.

[3] D. J. Bernstein. Cache-timing attacks on AES, 2005.

[4] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5):701–716, 2005.

[5] M. Hamburg. Accelerating AES with vector permute instructions. In *Workshop on Cryptographic Hardware and Embedded Systems*, 2009.

[6] A. Langley. ctgrind: Checking that functions are constant time with Valgrind. `https://github.com/agl/ctgrind`, 2010.

[7] A. Langley. Lucky Thirteen attack on TLS CBC. `https://www.imperialviolet.org/2013/02/04/luckythirteen.html`, 2013.

[8] J. Maurer. N4314: Data-invariant functions. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4314.html`, 2014.

[9] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *USENIX Annual Technical Conference, General Track*, pages 17–30, 2005.

[10] A. S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, 2001.

[11] V. Vasiliev. Decor: a DSL secure against timing attacks. `https://github.com/vasilvv/decor/blob/master/SUMMARY.md`, 2014.

[12] Y. Yarom and K. E. Falkner. Flush+ reload: a high resolution, low noise, l3 cache side-channel attack. *IACR Cryptology ePrint Archive*, 2013:448, 2013.

APPENDIX
A. INSTRUCTION CHECKER

We have developed a patch against Valgrind which triggers a "use of undefined value" error when one of the operands of multiplication is a private value. The patch adds the following code into multiple functions in Valgrind's memory checker:

```
switch (op) {
  case Iop_DivS32:
  case Iop_DivU32:
  case Iop_DivU32E:
  case Iop_DivS32E:
  case Iop_DivS64:
  case Iop_DivU64:
  case Iop_DivS64E:
  case Iop_DivU64E:
  case Iop_DivModU64to32:
  case Iop_DivModS64to32:
  case Iop_DivModU128to64:
  case Iop_DivModS128to64:
  case Iop_DivModS64to64:
  case Iop_Mul64:
  case Iop_Mul32:
  case Iop_Mul16:
  case Iop_Mul8:
    complainIfUndefined(mce, atom1, NULL)
    complainIfUndefined(mce, atom2, NULL)
  default:
    break;
}
```

The first problem here is that upstreaming this patch is problematic, since memcheck does not need those cases in order to detect undefined behavior. The second problem is that we have to take the most inclusive list of instructions which leak information through timing on various platforms, and map all of them onto the list of the instructions used by Valgrind's internal machine. We believe that both of those are feasible, but they would require us to actively communicate with the upstream.