Autocorrelation-Based Detection of Infinite Loops at Runtime

Andreas Ibing, Julian Kirsch and Lorenz Panny
Chair for IT Security
TU München
Boltzmannstrasse 3, 85748 Garching, Germany

***Abstract*—We present a new algorithm for the detection of infinite loop bugs in software. Source code is not needed. The algorithm is based on autocorrelation of a program execution's branch target address sequence. We describe the implementation of the algorithm in a dynamic binary instrumentation tool; the result is lightweight enough to be applied continuously at runtime. Functionality of the tool is evaluated with infinite loop bug test cases from the Juliet test suite for program analyzers. Applicability of the algorithm to production software is demonstrated by using the tool to detect previously known infinite loop bugs in `cgit`, `Avahi` and `PHP`.**

***Keywords*-Program analysis; infinite loops; dynamic binary instrumentation**

I. Introduction

Infinite loop bugs are both an issue of software safety and of software security. A program that runs into an infinite loop becomes unresponsive, which violates safety properties. If an infinite loop can be triggered with program input by an attacker, the program is vulnerable to a denial of service attack. Under the common weakness enumeration [1], infinite loops are known as CWE-835 ('loop with unreachable exit condition').

Possible mitigation approaches to infinite loop bugs include the usage of testing tools in order to detect and remove bugs before deployment, or to use light-weight program instrumentation to catch an infinite loop when in occurs at runtime.

The problem of finding infinite loops in programs is as old as computing itself. A perfect bug checker would detect all infinite loops in any program without false negative detections, without false positive detections, and with bounded runtime. Because the Halting Problem is known to be undecidable in general [2], such a perfect bug checker can not exist. Any real detection algorithm must therefore drop at least one of the three properties and tolerate either false positive detections, false negative detections, or running into non-termination itself.

Nevertheless, three principle approaches to automated infinite loop detection have been developed and are in practical use today. The oldest one is to set a maximum duration after which an unresponsive program is assumed to be stuck in an infinite loop. This approach is most often implemented with a watchdog timer in embedded systems: A watchdog timer has to be actively reset by the software perpetually, otherwise the watchdog triggers a system restart. A more recent example is found in dynamic symbolic execution tools [3], [4]. Dynamic symbolic execution is a testing approach where program execution is automatically driven into different program paths. The corresponding program input is automatically generated with a constraint solver, typically a Satisfiability Modulo Theories (SMT, [5]) solver. These symbolic execution tools also set a maximum duration of unresponsiveness, after which an infinite loop bug is reported.

The second approach is to prove termination or non-termination using an automated theorem prover, again typically an SMT solver. While (non-)termination can not be decided for all loops in general, it can be decided often enough to be of practical use. Prominent tools include [6]–[11]. Because these tools are not perfect, there is still an interest in detecting infinite loops at runtime. The process of translating a program into logic equations and especially theorem proving itself is too complex to be applicable continuously at runtime. In [12] it is proposed to connect a symbolic execution tool called `Looper` at a user's request to an unresponsive process. The tool would then single-step with symbolic execution and try to prove that an infinite loop is executed.

The third approach uses hash values of concrete program states to verify the execution of an infinite loop by finding a recurrence program state [13]. The tool presented in [13] is connected at a user's request to an unresponsive process and single-steps this process. At branches, the program state including register and memory contents is hashed and compared to previous hash values. If a hash appears more than once, an infinite loop is reported.

This paper presents a new algorithm for automated detection of infinite loops. It is sufficiently lightweight to be applied continuously at runtime. It does not need constraint solver, program source code or hash values over program states. The algorithm is based on *autocorrelation*, a computation commonly used in many areas of applied statistics: One example is in time series analysis to identify periodic changes. Another example is in signal processing,

```c
int main(int argc, char **argv) {
    unsigned int i, j, k = 0;
    /* Infinite loop: increment i instead of j */
    for (j = 1; j < 0x10000; i++)
        for (i = 0; i < 0x10; i++)
            k++;
}
```

Figure 1: Sample C program containing an infinite loop

```
004004b6 <main>:
004004b6:       push   rbp
004004b7:       mov    rbp,rsp

004004ba:       mov    DWORD PTR [rbp-0x14],edi
004004bd:       mov    QWORD PTR [rbp-0x20],rsi
004004c1:       mov    DWORD PTR [rbp-0x8],0x1
004004c8:       jmp    4004e5 <main+0x2f>
004004ca:       mov    DWORD PTR [rbp-0x4],0x0
004004d1:       jmp    4004db <main+0x25>

004004d3:       add    DWORD PTR [rbp-0xc],0x1
004004d7:       add    DWORD PTR [rbp-0x4],0x1
004004db:       cmp    DWORD PTR [rbp-0x4],0xf
004004df:       jbe    4004d3 <main+0x1d>
004004d7:       add    DWORD PTR [rbp-0x4],0x1

004004e5:       cmp    DWORD PTR [rbp-0x8],0xffff
004004ec:       jbe    4004ca <main+0x14>
004004ee:       mov    eax,0x0
004004f3:       pop    rbp
004004f4:       ret
```

Figure 2: Compiled version of the sample program

for synchronization in communication systems. We apply autocorrelation as an efficient way to identify suspicious branch sequences on-the-fly.

The remainder of this paper is organized as follows. Section II describes and illustrates the algorithm in detail. Section III presents an implementation using dynamic binary instrumentation. Algorithm properties are described in detail in Section IV. Experiments with the resulting tool are described in section V. The tool is evaluated with infinite loop test cases from the Juliet suite [14], and it is used to detect known infinite loop bugs in `Avahi`, `cgit` and `PHP`. Related work is reviewed in section VI. Results of the tool evaluation are discussed in section VII.

II. Algorithm: Modified Autocorrelation for Branch Target Address Sequence Analysis

Autocorrelation is the correlation of a function $f(n)$ with itself at different points in time. It computes the similarity between the function and a time-lagged version of itself, where the time lag l is variable. The (discrete) autocorrelation R_{ff} at lag l for a real-valued function $f(n)$ is:

$$R_{ff}(l) = \sum_{n \in \mathbb{Z}} f(n) f(n-l)$$

For a periodic function $f(n)$ we have:

$$f(n+p) = f(n)$$

where p is the period length. The autocorrelation function $R_{ff}(l)$ always has a peak value for $l = 0$. For a periodic function of period p, the autocorrelation also peaks at the period length $l = p$ and its integer multiples. For the detection of infinite loops in programs, we slightly adapt the autocorrelation as described in the following sections.

A. Detecting periodic infinite loops

A path through a program can be represented by the corresponding sequence of branches, i.e., the sequence of branch target addresses. The branches comprise both conditional branches and unconditional branches (jumps). A branch target address sequence is denoted as:

$$b(n) \quad n \in [0, m]$$

where $b(0)$ is the first branch after the program's entry point, and $b(m)$ is the last taken branch, leading to the current program execution state. The basic idea is to detect an infinite loop through the branch sequence, which is assumed to become periodic. To this end, the autocorrelation of the branch sequence is computed on-the-fly during program execution.

$$R_{bb}(l, m) = \sum_{n=1}^{m} b(n) b(n-l) \ , \quad l = 0..m$$

The value for $l = 0$ is not of interest and not computed, because an infinite loop has a positive period length. If the program runs into an infinite loop with period length p, the autocorrelation will show a peak at $R_{bb}(l = p, m)$, where the peak value increases with the number of branches m.

Unlike in other applications of autocorrelation where a periodic function undergoes some time variance due to noise, here we have identical periods. Therefore, we replace the multiplication with Kronecker's delta function:

$$\delta(i, j) = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{else} \end{cases} \qquad i, j \in \mathbb{Z}$$

That yields:

$$R_{bb}(l, m) = \sum_{n=1}^{m} \delta\Big(b(n), b(n-l)\Big) \ , \quad l = 0..m$$

i.e., the autocorrelation value is only increased if the branch target address is identical to the target address of l branches before. The recursive version for on-the-fly computation is:

$$R_{bb}(l, m) = R_{bb}(l, m-1) + \delta\Big(b(m), b(m-l)\Big)$$

We also want to consider the falsification of an infinite loop hypothesis. If a periodic sequence is broken, we reset the corresponding correlation value(s). The recursive computation becomes:

$$R_{bb}(l, m) = \begin{cases} R_{bb}(l, m-1) + 1 & \text{if } b(m) = b(m-l) \\ 0 & \text{else} \end{cases} \tag{1}$$

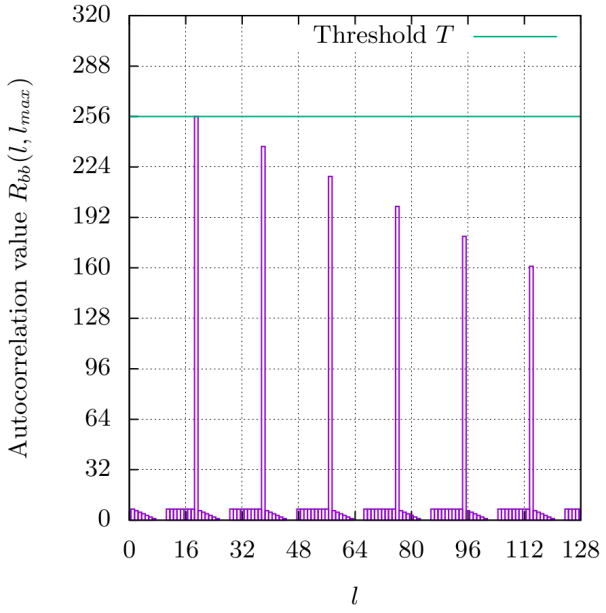

Figure 3: A static snapshot of the autocorrelation values $R_{bb}(l, l_{max})$ at forcibly induced program termination ($R_{bb}(18, l_{max}) \geq T$)

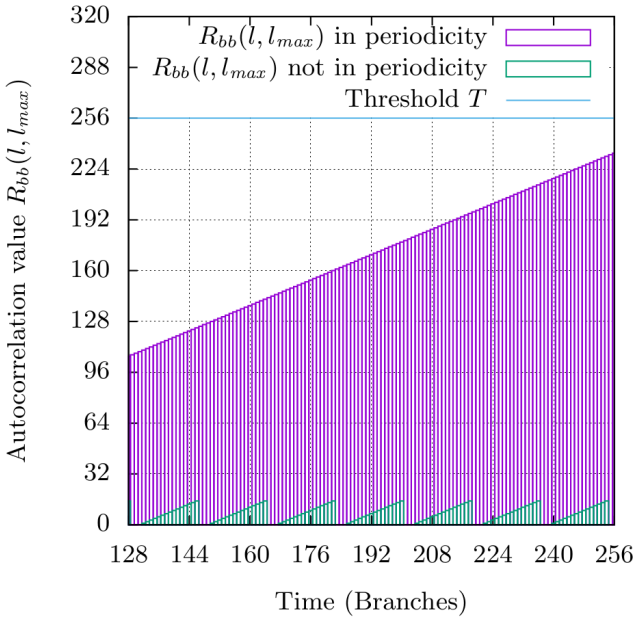

Figure 4: A selected time frame observing two dynamically changing values ($R_{bb}(18, l_{max})$, $R_{bb}(19, l_{max})$) during program execution

An infinite loop candidate is identified if the correlation exceeds a pre-defined threshold value T:

$$R_{bb}(l, m) > T \quad \text{for any } l \tag{2}$$

the period of the infinite loop is $p = l$, i.e., the index of the correlation value that first exceeds the threshold T. The location of the infinite loop is given by the p last branch target addresses.

Correlation length: In order to limit the buffer length for branch target addresses and autocorrelation values to a constant size independent of the length of a program path, the autocorrelation length can be limited. $R_{bb}(l, m)$ is then only computed for indices $l \in [1..l_{max}]$. With such a fixed correlation length, the algorithm detects infinite loops with a period of up to $p = l_{max}$.

Illustration: We provide a sample program as depicted in Figure 1 to clarify how the algorithm behaves during program execution. To understand the results, we briefly consider the x86 assembly code (Figure 2 generated out of 1 by the GNU C compiler (version 4.9.2): The outer (infinite) loop is transformed into an unconditional branch at virtual address `0x4004c8` and a conditional one at address `0x4004ec`. Similarly, the branches at `0x4004d1` and `0x4004df` represent the inner loop. Local variables `i`, `j` and `k` are addressed relatively to the base pointer `rbp` at offsets `-0x4`, `-0x8` and `-0xc` within the current stack frame. Figure 3 shows the autocorrelation values at the time of (forced) program termination (that is $\exists l\colon\ R_{bb}(l, l_{max}) \geq T$) when choosing $l_{max} = T = 2^8$. The peaks at positions p which are non-zero multiples of 19 clearly indicate the periodicity of the generated assembly code: To perform one complete iteration of the outer loop, the conditional branch at `0x4004df` corresponding to the inner loop is considered 17 times (16 times taken, once not taken) by the algorithm, and with the conditional jump at `0x4004ec` and the unconditional jump at `0x4004d1` this results in a total of 19 branches. The diagram in Figure 4 visualizes the behaviour of two selected autocorrelation values while the instrumented program is executing. We chose one autocorrelation value at a position congruent to the periodicity of the sample program and the direct neighbouring value and plot the different behaviours over time. The diagram shows that the first value increases steadily while the second autocorrelation value follows a sawtooth-like shape.

B. Detecting certain non-periodic infinite loops

There are non-periodic infinite loops, i.e., infinite loops whose branch sequence depends on input. An example is shown in Figure 5 in assembly. Every second branch targets `.loop`, but the branch in between is random.

The algorithm can be generalized to also detect infinte loops with non-deterministic branches, as long as the branch sequences in the loop have equal length. The following recursive correlation detects re-visited branches with constant

```
1 .loop:
2         rdrand %ax
3         test $1, %ax
4         jnz .one
5 .zero:
6         jmp .loop
7 .one:
8         jmp .loop
```

Figure 5: Example non-periodic infinite loop in x86 assembly

branch sequence length:

$$R_{bb}(l,f,m) = \begin{cases} R_{bb}(l,f,m-1) & \text{for } f \not\equiv m \pmod{l} \\ R_{bb}(l,f,m-1)+1 & \text{for } f \equiv m \pmod{l} \\ & \quad \wedge\, b(m) = b(m-l) \\ 0 & \text{for } f \equiv m \pmod{l} \\ & \quad \wedge\, b(m) \neq b(m-l) \end{cases} \tag{3}$$

where f is an offset modulo l. The coefficent vector becomes a triangular matrix (compare Figure 6).

The number of coefficient computations is the same as before, i.e., for every new branch and for each l, only one coefficient is updated (the one for which $f \equiv m \pmod{l}$). But the number of coefficients increases to:

$$N_{\text{coeff}} = \frac{l_{\max}}{2}(l_{\max}-1),$$

i.e., the space for coefficient storage increases quadratically with correlation length. The example non-periodic infinite loop is detected with the extended algorithm (triangular coefficient matrix) with $l_{\max} \geq 2$.

C. Avoiding false positives: optional SMT-based verification of candidate loops

Depending on the theshold value and the application, the presented algorithm has false positive detections. If false positives are not tolerable, there is a possibility for verification or falsification of infinite loop candidates. One iteration of the candidate loop is executed with symbolic execution, and an SMT solver is used to check whether the loop is indeed infinite. If the candidate is not verified, the threshold can be increased by some factor. The SMT check for an infinite loop is the same as described in [12] for the `LOOPER` tool. This optional candidate verification however differs from `LOOPER` in that `LOOPER` searches for an infinite loop with unknown period length and unknown location. Therefore, `LOOPER` needs a number of solver checks that increases exponentially with period length, while the work at hand only needs one check for the infinite loop candidate.

III. Implementation with Binary Instrumentation

We implemented our algorithm based on dynamic binary instrumentation engine *Pin* [15].

Dynamic binary instrumentation provides a mechanism to monitor, inspect and alter the execution of any given binary program at runtime. This is typically achieved by injecting callback functions into a Just-In-Time (JIT) compiled version of the instrumented program which can then observe and/or manipulate the internal state of the program. Dynamic binary instrumentation lends itself very well towards program analysis as it allows for fine-grained inspection of arbitrary code without having to rely on the presence of the executable's source code. Programs based on instrumentation frameworks are usually referred to as *tools*.

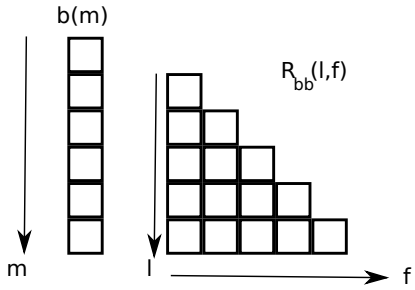

Figure 6: Triangular coefficient matrix

The source code of our implementation can be seen in Figure 7: The code adds a callback function `Branch()` to all conditional and unconditional branches that reside in the virtual address space of the main executable program image. It is the injected handler function's responsibility to re-compute each autocorrelation value R_{bb} (line 15) and to store the most recent branch target. Moreover, the `Branch()` function ensures that T is still the upper bound of all values residing in R_{bb}. In case of a violation of this last constraint, the Pin tool outputs a warning in line 18.

IV. Properties

A. Complexity

1) Time: The number of operations performed by the algorithm per branch depends on the correlation length $l_{\max}$. The algorithm updates $l_{\max}$ coefficients for every new branch, and compares them against the threshold. It can therefore be assumed, that the computation overhead increases linearly with correlation length l_{max}, and is independent of program length. To confirm this, we instrumented an infinite-loop-free (terminating) version of the sample program with our algorithm implementation as Pin tool (using equation (1), i.e., without the modulo addressing of equation (3)) and measured the total runtime for different values l_{max}. In the changed sample program, line 6 was updated to increment the local variable `j` instead of `i`. Benchmark results can be seen in Figure 8.

2) Space: The amount of memory required to store branch target addresses increases linearly with correlation length. The amount of memory required to store autocorrelation values increases linearly with correlation length when using equation (1), and quadratically when using equation (3).

```
#include "pin.H"                    /* All other includes
    omitted for readability reasons */

#define T           (500)          /* Threshold triggering
    an abort */
#define M           (16)           /* Number of values */
#define MASK(X)     ((X) % M)

size_t cur = 0;
unsigned long dst[M] = { 0 };      /* The m last branch
    targets */
unsigned long cnt[M] = { 0 };      /* The m values R_bb(l,
    m) */

void Branch(unsigned long ip, bool taken, unsigned long
    target, unsigned long fallthrough) {
    if (!taken) target = fallthrough;

    /* Check if T has been surpassed by any of the
        counters and update autocorrelation values */
    for (size_t i = 0; i < M; i++) {
        cnt[i] = (dst[MASK(cur - i)] == target) ? cnt[i] +
            1 : 0;
        if (cnt[MASK(cur - i)] >= T)
            error(-1, 0, "Infinite loop detected.
                Instruction: %p Target: %p\n", ip, target
                );
    }

    /* Store most recent branch target */
    cur = MASK(cur);
    dst[cur++] = target;
}

void Instruction(INS ins, void *v) {
    /* For any newly translated branch instruction in the
        main image, add a callback to Branch() */
    if (INS_Category(ins) == XED_CATEGORY_COND_BR ||
        INS_Category(ins) == XED_CATEGORY_UNCOND_BR) {
        IMG img = IMG_FindByAddress(INS_Address(ins));
        if (IMG_Valid(img) && IMG_IsMainExecutable(img))
            INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)
                Branch, IARG_INST_PTR, IARG_BRANCH_TAKEN,
                    IARG_BRANCH_TARGET_ADDR,
                        IARG_FALLTHROUGH_ADDR,
                        IARG_END);
    }
}

int main(int argc, char **argv) {
    if (PIN_Init(argc, argv)) return -1;
    INS_AddInstrumentFunction(Instruction, 0);
    PIN_StartProgram();
}
```

Figure 7: Implementation of the algorithm based on the *PIN* binary instrumentation framework

3) Runtime Overhead: The runtime overhead of the Pin based instrumentation consists of three parts. A one-time overhead independent of program length and a dynamic constant overhead for every branch are due to Pin. The third part is the actual autocorrelation, that causes a constant overhead for every branch, where the overhead increases linearly with correlation length (using equation (1)).

The Intel Pin creators estimate the runtime overhead added by the instrumentation engine at a factor of 2.8 with just the JIT compiler enabled and at an average factor of 7.8 for a basic-block counting instrumentation tool in the worst case [15]. Figure 8 shows a run time of about 2 ms for the modified (terminating) test program running without instrumentation. The same program takes about 200 ms running within an empty Pin instance. We further obtain a runtime of about 450 ms for the sample program running within Pin using a correlation length of 100. This implies a slowdown of factor $\frac{450}{2} = 225$ for the example correlation length.

If faster execution is desirable, we consider to implement the described algorithm as compiler instrumentation (static binary instrumentation at compile time). This gets rid of the overhead introduced by Pin. The overhead added by Pin are the fixed one-time overhead introduced by Pin's JIT as well as the dynamic part that depends on the number of branches within the target program. Both are independent of the correlation length l_{max}. We thus argue that a compiler based instrumentation shows the same slope in dependence on l_{max}. This runtime is depicted as a dashed (green) line in Figure 8, below the blue linear regression curve for measured values with Pin instrumentation. The blue line is the graph of the linear function $t(l_{max}) = 2156.2 \cdot l_{max} + 302700$ (t in milliseconds). For $l_{max} = 100$ the slowdown factor decreases to about 100 by changing from dynamic instrumentation to static instrumentation. Additionally, our example program consists of an unnaturally high fraction of branches compared to control-flow presevering instructions of $\approx \frac{1}{3}$. For real-world programs, one would expect this fraction between 10% and 20%. This would decrease the slowdown factor again, resulting in a slowdown factor of about 50 to 70 for real-world programs with compiler instrumentation and correlation length 100.

B. Incompleteness

Completeness would mean the detection of all infinite loops, i.e., that there are no false negative detections. Since there are non-periodic infinite loops that are not detected by the presented algorithm, and since periodic infinite loops with periodicity larger than correlation length are not detected, the presented algorithm is not complete.

C. Soundness

Soundness means that any loop which is reported as infinite by the algorithm is indeed infinite, i.e., that there are no false positive detections. The presented algorithm uses a threshold comparison to select infinite loop candidates. This candidate selection is not sound. It could arguably be seen as 'asymptotically sound' with an increasing threshold value. The algorithm becomes sound by applying an SMT check for a candidate and by only reporting loops whose non-termination was proven by the check.

D. Trade-offs

The tool user can set the two algorithm paramters l_{max} for correlation length and T as detection threshold. There are two trade-offs. The first trade-off concerns the number of false negative candidate detections versus algorithm

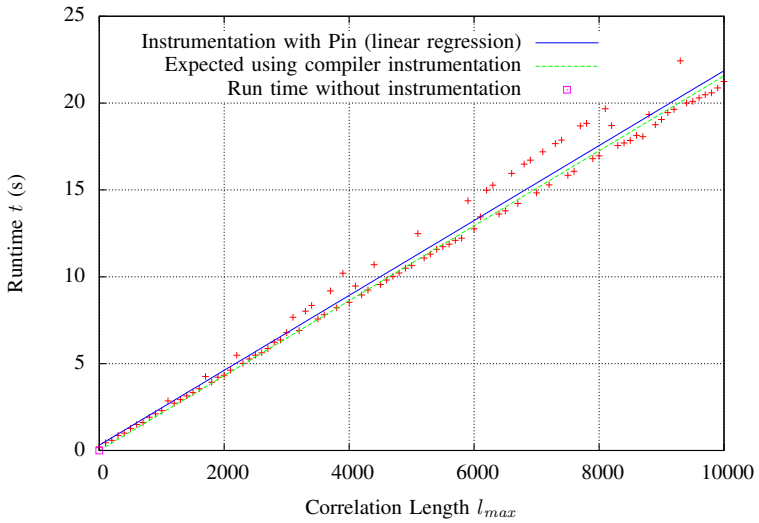

Figure 8: Runtimes of the algorithm applied to a modified (terminating) version of the sample program.

complexity. By increasing correlation length, the number of false negative candidate detections is reduced at the expense of increased program overhead. The second trade-off concerns the number of false positive detections versus detection delay. By increasing the threshold T, the number of false positive candidate detections is reduced at the expense of increased detection delay. An infinite loop must then be executed for more iterations until the threshold is reached. In our practical tests, the threshold could be set very high (10^6 or more) without causing any significant detection delay.

V. Experiments

Benchmark results are obtained on an Intel Core i7-2600 (3.4 GHz) on 64bit Linux kernel 3.16.0-4-amd64. The algorithm is run with equation (1).

A. Juliet suite

The Juliet test suite [14] is currently the most comprehensive test suite for C/C++ program analyzers. It is developed by the United States National Security Agency's (NSA) Center for Assured Software and the National Institute of Standards (NIST). The suite is freely available and consists of C/C++ source code covering over 100 common weaknesses. The test cases are artificial and systematically combine basic bugs with different control and data flow variants. We use the current Juliet version 1.2. For infinite loops it contains 6 test programs, each with 'good' (bug-free) and 'bad' functions (containing a bug).

An example is shown in Figure 9. The infinite loop has a period of 1. The terminating 'good' loop is breaked after 10 iterations.

We compile the programs and apply our tool with parameters $T = 500$ and $l_{max} = 16$. The tool correctly detects the contained infinite loops without false positive and without false negative detections. The runtimes until detection of the respective infinite loop are given in Table I.

The Juliet infinite loop tests have also been used in [16] to test detection based on static symbolic execution on the source code level. An SMT solver is used to check for fixed-point satisfiability. Compared to [16], our tool achieves correct detection an order of magnitude faster. For bigger test programs, we would expect a larger speed-up with our tool, because on the small Juliet test programs a considerable portion of the runtime is used to perform the instrumentation.

B. Real-world programs

To show that the algorithm scales to real-world problems, we triggered publicly known infinite-loop vulnerabilities in three widely-used open-source programs and successfully detected all of them using our implementation. The three vulnerabilities outlined below have been selected based on the ease of triggering the bug.

- *cgit* is a web frontend for git repositories written in C.[1] Prior to version 0.8.3.5, cgit contained an infinite loop bug that could be triggered by clients sending an invalid hex escape in the URL query to the remote side. This bug has been assigned CVE-2011-1027.
- *Avahi* is a Unix daemon providing service discovery in local networks which is enabled by default in many popular Linux distributions.[2] Before version 0.6.29, Avahi would enter an infinite loop upon receiving a zero-length UDP packet (CVE-2011-1002).
- *PHP: Hypertext Processor* is a highly popular server-side scripting language for web development.[3] All versions before 5.2.17/5.3.5 hang infinitely when processing

[1] http://git.zx2c4.com/cgit/
[2] http://avahi.org/
[3] http://php.net/

```
void do_01_good1() {
    int i = 0;
    do {
        if (i == 10) {
            break;
        }
        printIntLine(i);
        i = (i + 1) % 256;
    } while (i >= 0);
}

void do_01_bad() {
    int i = 0;
    do {
        printIntLine(i);
        i = (i + 1) % 256;
    } while(i >= 0);
}
```

Figure 9: Example from Juliet test suite [14]

the string representation of the floating-point value $2.2250738585072011 \cdot 10^{-308}$. This infinite loop bug is referenced as CVE-2010-4645.

The results of our evaluation are compiled in Table II. It features the detected jump target address period length (as defined by the l value which caused $R_{bb}(l, l_{max})$ to exceed the threshold T) as well as the wall-clock time difference between triggering the bug and forced termination of the instrumented program using our implementation of the algorithm.

In all three cases, we used the threshold value $T = 1024$ and initially conducted tests with $l_{max} = 64$. This yielded positive results for the cgit and Avahi tests, detecting the infinite loops of periodicities 22 and 3 within 0.23 and 0.07 seconds, respectively. We discovered $l_{max} = 64$ to be too few for the PHP test, since that bug exhibits a period length of 69 branches: After increasing l_{max} to 128, the infinite loop was detected within 0.78 seconds.

VI. Related Work

Previous work on infinite loop detection relies on a threshold for a maximum duration of unresponsiveness (e.g., [3], [4]), on automated theorem proving [7]–[12], [16], or on comparing program states at branches [13]. The presented algorithm clearly differs from these approaches.

On the one hand, specifying a threshold for maximum number of loop iterations does bear some resemblance to specifying a maximum duration of unresponsiveness. On the other hand, it is not straight-forward to predict an adequate time threshold, as execution speed depends on many parameters like the underlying hardware, program input or system load. This is why watchdogs are mainly used in embedded systems, where these parameters are predictable. An iteration threshold is independent of such parameters, and could additionally be inferred from program source code if available (to avoid false detections).

The main difference to the theorem proving approach is that it reasons about a multitude of program paths at once, while the proposed algorithm is applied to an individual program path. The prominent tools use static symbolic execution of the program source code, with loop invariant generation [7], [8] and/or checking for satisfiable fixed-points or more generally recurrence sets [7], [11]. Proving non-termination is sound, i.e., it does not yield any false positive infinite loop detections. In `Looper` [12], it is proposed to use a theorem prover to verify that a program is stuck in an infinite loop. The tool is to be run at user's request, and single-steps the unresponsive program with symbolic execution. The work at hand for detection of infinite loop candidates with autocorrelation does not need a solver, and is orthogonal to the theorem proving approach. The presented algorithm spots unresponsiveness automatically by checking the iteration number threshold. The theorem proving approach is much more complex, which makes it suited for analysis before deployment. The presented algorithm on the other hand is lightweight enough for detection at runtime.

Testcase	Runtime (s)
`Infinite_Loop__do_01`	0.20
`Infinite_Loop__do_true_01`	0.20
`Infinite_Loop__for_01`	0.14
`Infinite_Loop__for_empty_01`	0.19
`Infinite_Loop__while_01`	0.14
`Infinite_Loop__while_true_01`	0.14

Table I: Error detection runtimes of the infinite loop test cases from the Juliet suite

The work at hand also differs from the approach of finding concrete fixed-points by comparing program states at branches, as proposed in [13]. Program states can become quite big, which makes this approach too complex to be run continuously. In [13], it is proposed to apply program state comparisons only on demand in case a process becomes unresponsive. As noted in [13], this approach misses infinite loops where program states are not identical. Comparing process states as described in [13] is also not sound, i.e., it might false positively report an infinite loop, because there is also a kernel state (packet queues etc.) for a running process. The kernel state is not included in the state comparisons.

VII. Discussion

Unlike previous work for sound detection, the presented algorithm is lightweight enough for continuous detection of infinite loops at runtime. While a watchdog timer needs to be actively used by the programmer, the presented algortihm can be used to monitor any program.

In order to reduce the average overhead introduced by the presented infinite loop detection, the correlation length could be adaptively changed during program execution. One possibility is to introduce a lower-complexity 'search mode', where no correlation is computed at all. Instead, counters can be increased for how often the program execution

CVE	Name	l_{max}	T	Detection Time	Measured Periodicity
CVE-2011-1027	cgit	64	1024	0.23 s	22
CVE-2011-1002	Avahi	64	1024	0.07 s	3
CVE-2010-4645	PHP	128	1024	0.78 s	69

Table II: Selected CVEs

passes individual branch locations. This essentially reduces the algorithm overhead to an overhead value required by instrumentation to trace code coverage, like, e.g., `gcov`. If any branch counter exceeds a pre-defined threshold T_s, the algorithm can change into 'correlation mode'.

There is the possibility of integrating the presented algorithm in CPUs, in order to benefit from hardware acceleration. Most CPUs already have a branch target address cache. Detection of an infinite loop could be signaled with an interrupt (to the operating system) and an operating system signal (to a process). The operating system could perform verification / falsification with an SMT check to eliminate false positives. Hardware acceleration would eliminate the runtime software overhead of the proposed algorithm for correlation at the expense of some extra transistors.

Future work could be the re-implementation of the algorithm with static binary instrumentation, and integration into the GNU compiler (`gcc`). `gcc` already features an address sanitizer [17] and a thread sanitizer [18], but not yet a loop sanitizer.

It seems further possible to combine the presented algorithm with dynamic symbolic execution based testing. The synergy would be that symbolic execution drives the program into different paths, where the presented algorithm could perform lightweight detection of infinite loop candidates.

References

[1] R. Martin, S. Barnum, and S. Christey, "Being explicit about security weaknesses," *CrossTalk The Journal of Defense Software Engineering*, vol. 20, pp. 4–8, 3 2007.

[2] A. Turing, "On computable numbers, with an application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 42, pp. 230–265, 1937.

[3] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.

[4] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *European Software Engineering Conference and International Symposium on Foundations of Software Engineering*, 2005, pp. 263–272.

[5] L. deMoura and N. Bjorner, "Satisfiability modulo theories: Introduction and applications," *Communications of the ACM*, vol. 54, no. 9, 2011.

[6] A. Podelski and A. Rybalchenko, "A complete method for the synthesis of linear ranking functions," in *Int. Conf. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2004.

[7] A. Gupta, A. Rybalchenko, T. Henzinger, R. Xu, and R. Majumdar, "Proving non-termination," in *Symp. Principles of Programming Languages (POPL)*, 2008.

[8] H. Velroyen and P. Rummer, "Non-termination checking for imperative programs," in *Tests and Proofs (TAP)*, 2008.

[9] E. Payet and F. Spoto, "Experiments with non-termination analysis for Java Bytecode," in *BYTECODE*, 2009, pp. 83–96.

[10] M. Brockschmidt, T. Ströder, C. Otto, and J. Giesl, "Automated detection of non-termination and NullPointerExceptions for Java Bytecode," in *Int. Conf. Formal Verification of Object-Oriented Software*, 2011, pp. 123–141.

[11] H. Chen, B. Cook, C. Fuhs, K. Nimkar, and P. O'Hearn, "Proving nontermination via safety," in *Tools and Algorithms for the Construction and Analysis of Systems*, 2014, pp. 156–171.

[12] J. Burnim, N. Jalbert, C. Stergiou, and K. Sen, "Looper: Lightweight detection of infinite loops at runtime," in *Int. Conf. Automated Software Engineering*, 2009.

[13] M. Carbin, S. Misailovic, M. Kling, and M. Rinard, "Detecting and escaping infinite loops with Jolt," in *European Conf. Object-Oriented Programming*, 2011, pp. 609–633.

[14] T. Boland and P. Black, "Juliet 1.1 C/C++ and Java test suite," *IEEE Computer*, vol. 45, no. 10, 2012.

[15] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proc. ACM Conf. Programming Language Design and Implementation*, 2005, pp. 190–200.

[16] A. Ibing and A. Mai, "A fixed-point algorithm for automated static detection of infinite loops," in *IEEE Int. Symp. High Assurance Systems Eng.*, 2015, pp. 44–51.

[17] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *USENIX Annual Technical Conference*, 2012, pp. 28–28.

[18] K. Serebryany and T. Iskhodzhanov, "ThreadSanitizer: data race detection in practice," in *Workshop on Binary Instrumentation and Applications*, 2009, pp. 62–71.