UCTA: Dynamic Program Analysis on Instruction Traces

Kevin Laeufer
RWTH Aachen University
kevin.laeufer@rwth-aachen.de

Abstract

We present UCTA, a framework for dynamic program analysis on instruction traces from low power embedded microcontrollers. Our tool allows to perform analysis similar to Valgrind and AddressSanitizer without instrumentation and thus without altering the timing behavior. We are able to run our analysis without a memory trace by recreating the missing memory state through concrete execution of the recorded sequence of instructions. A callback based interface makes it easy to implement various dynamic analysis algorithms on top of our engine. As a proof of concept we have implemented a dynamic out-of-bounds detection for stack based arrays.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging

Keywords microcontroller; instruction trace; dynamic binary analysis; shadow values; buffer overflow

1. Introduction

No matter if they are part of the Internet of Things (IoT), Industry 4.0 or Cyber Pysical Systems, more and more traditional embedded systems are connected to the internet and thus exposed to various forms of malicious user input. Recently researchers were able to access the local CAN network of a car through its cellphone modem [8]. Another way of sending arbitrary input to IoT devices is to inject malicious packets into a wireless automation network as shown by [7].

While these IoT devices might soon be the target of attacks very similar to the ones common on servers and personal computers, their hardware features, development tools and practices are very different: While today a lot of web facing software is written in managed languages, microcontrollers are still predominately programmed in C and C++ which place the burden of managing memory soly on the programmer. In addition, there is no support for virtual memory and thus no address space layout randomization. While simple memory protection units are available, the most common IoT operating systems, Contiki [4] and RIOT-OS [1] still operate in one common address space.

As there is no protection provided by the programming language and no reliable mechanisms to detect and contain a memory bug, software needs to be tested to find and repair any critical bugs before shipping a product that will be connected to the internet.

For C and C++ programs on a regular computer, Valgrind [9] and AddressSanitizer [11] are widely used tool that can detect memory errors as a program is executed. Both of them instrument the program and use shadow memory to track which memory regions can be accessed by the program and to check these invariances. Thus they incur a timing overhead (about 20x for Valgrind, 2x for AddressSanitizer) as well as a higher memory consumption. While the overhead is acceptable for testing on a computer, microcontroller firmware is often times very timing critical. A 2x slowdown of code in an important interrupt routine could mean that a program no longer works. In addition, these low power embedded devices are highly memory constrained such that there might not be enough RAM left to use as shadow memory.

Some modern microcontrollers are capable of providing information on the instructions that they recently executed. This information along with a copy of the firmware image is enough to recreate the sequence of instructions that were executed on the embedded processor. However, for the low power microcontroller there is no information about the results of memory reads available. Commercial tools such as the Lauterbach TRACE32[1] application and associated hardware are able to sample this information for a microprocessor running at full speed. However, their analysis of the data mostly focuses on providing information to a firmware developer such as profiling information and an interactive view of the instructions stream. They do not provide any advanced automated program analysis that would be able to uncover bugs without the input from a developer.

[1] `http://www.lauterbach.com`

We are the first to combine dynamic analysis techniques similar to the ones used in the Memcheck and AddressSanitizer tools with instruction traces in order to automatically detect bugs in microcontroller firmware, once they were executed. This is a first step to automated testing of firmware on these devices through fuzzing or concolic testing. In this paper we discuss UCTA, a framework on top of which various heavy weight dynamic program analysis tools can be implemented. We then present proof of concept implementations for three different analysis passes with increasing complexity.

2. Microcontroller Trace Analysis (UCTA)

In this section we present UCTA, a framework for dynamic program analysis on instruction traces. We define how we simulate various kinds of memories, describe our execution core and the intermediate language that we use.

2.1 Memory Simulation

For our analysis we assume that the program under test accesses three different memory regions: (1) RAM which contains undefined values on startup (2) ROM which contains the firmware image and (3) memory mapped I/O. In order to know the contents of RAM, we require that the instruction trace contain all instructions starting with the first instruction that is executed after reset. This allows us to recreate the state of RAM. For ROM we just load the firmware image into memory. We currently just return zero when the program tries to read from memory mapped I/O. In the future we want to provide a simple interface to provide simulations of memory mapped peripherals that are relevant to the firmware under test. As we know which branches were taken in the original execution, imprecision in our memory model can only lead to false data values.

2.2 Shadow and Concrete Values

To support various kinds of analysis on top of a common core, our execution engine operates on tuples that contain the concrete value as well as shadow values associated with that value. In our pseudo code, the concrete value is accessed through a call to `concrete(value)`, while the shadow values are accessed through the `[]` operator by key: `value[key]`.

Our execution engine is stack based. Shadow values are propagated through value copies: when loading a value from a register or a memory location its concrete value as well as its associated shadow values are copied onto the stack. If a value is changed, for example, when a unary operation such as a bit-wise negation is performed on the data, the result of that operation will be treated as a new value tuple. If an analysis wants shadow values from the inputs of an operation to be propagated to the resulting value it needs to implement the corresponding callbacks to copy the values.

```
stack = []
for token in esil.split(','):
  if token in esil_commands:
    esil_commands[token](token, stack)
  else:
    stack.append(token)
```

Figure 1. ESIL interpreter core.

2.3 Radare2 and ESIL

To load and disassemble the firmware image, we rely on the open source `radare2` disassembler[2]. In addition to the traditional instruction mnemonic, `radare2` can emit a small program in a simple intermediate language that is semantically equivalent to the original instruction, but much easier to interpret. We make use of the Evaluable Strings Intermediate Language (ESIL) in order to keep the core of our analysis framework simple and largely target platform independent.

An ESIL program consists of comma separated instructions and literals. Instructions operate on memory, registers and values on the stack. Literals are pushed onto the stack and are eventually consumed by an instruction. The core of our ESIL interpreter is shown in Figure 1. The semantics of a simple subset of ESIL that we implement in our execution engine are shown in Figure 2. All other ESIL operations with more complex semantics can be represented by a series of these simpler operations. These include operations that load a value from a register, apply an operation to it and then store the result in the same register, as well as complex store and load operations. Figure 3 provides rewrite rules that define the mapping to the simpler subset of ESIL.

2.4 Analysis Tool Interface

In order to allow others to implement flexible and composable analysis tools on top of UCTA, we provide callbacks for relevant events in the execution. The mapping of callbacks to simple ESIL instructions is shown in Figure 2. The callback based interface was inspired by the design of the Jalangi JavaScript analysis framework [10].

For unary and binary operations, the type of operation performed is provided through the `src` shadow value of the result. Analysis tools are free to modify shadow values, but are encouraged to use a unique key for their operation. The order in which analysis tools are called is specified by the user. As all tools operate on common shadow values, information from low level analysis tools can be used by more high level analysis tools.

3. Analysis

In this section we present three different analysis tools that we implemented on top of the UCTA framework.

[2] `https://github.com/radare/radare2`

Description	ESIL	Stack			Operation	Callback
store to register	`=`	val, reg	→	ϵ	`R[reg] = val`	`on_store_to_reg(val, reg)`
store to memory	`=[`b`]`	val, addr	→	ϵ	`mem.store(addr, val, b)`	`on_store(val, addr)`
load from memory	`[`b`]`	addr	→	val	`val = mem.load(addr, b)`	`on_load(val, addr)`
increment	`++`	a	→	res	`res = a + 1`	
decrement	`--`	a	→	res	`res = a - 1`	`on_unary_op(a, c)`
negate	`!`	a	→	res	`res = 1 if a == 0 else 0`	
logical left shift	`<<`	b,a	→	res	`c = a << b`	
logical right shift	`>>`	b,a	→	res	`c = a lrs b`	
rotate left	`<<<`	b,a	→	res	`c = (a<<b)|(a>>(ws-b))`	
rotate right	`>>>`	b,a	→	res	`c = (a>>b)|(a<<(ws-b))`	
bitwise AND	`&`	b,a	→	res	`c = a & b`	`on_binary_op(a, c)`
bitwise OR	`|`	b,a	→	res	`c = a | b`	
bitwise XOR	`^`	b,a	→	res	`c = a ^ b`	**Overflow Behavior**
add	`+`	b,a	→	res	`c = a + b`	`res = int(c) & ((1<<32)-1)`
subtract	`-`	b,a	→	res	`c = a - b`	
multiply	`*`	b,a	→	res	`c = a * b`	
divide	`/`	b,a	→	res	`c = a / b`	
modulo operation	`%`	b,a	→	res	`c = a % b`	
equal	`==`	b,a	→	res	`res = 1 if a == b else 0`	
greater	`>`	b,a	→	res	`res = 1 if a >  b else 0`	
greater or equal	`>=`	b,a	→	res	`res = 1 if a >= b else 0`	`on_compare(a, b, c)`
smaller	`<`	b,a	→	res	`res = 1 if a <  b else 0`	
smaller or equal	`<=`	b,a	→	res	`res = 1 if a <= b else 0`	

Definition: number of bytes: $b \in \{1, 2, 4, 8\}$; word size: $ws = 32$

Literals are pushed on the stack. Once they are retrieved by an operation, their value is computed. Integer literals are converted to their corresponding value, register literals are reduced to their register value when the operation is evaluated. The only exception to this rule is the store to register operation (`=`) which uses the register as destination of the store.

Description	ESIL	Operation	Callback
load from register	`r[0-15]`, `ip`, `sp`, `lr`, `pc`	`val = R[reg]`	`on_load_from_reg(val, reg)`

Figure 2. Simple ESIL semantics on a 32 bit architecture and associated callbacks.

Compound Operations

$$\{src\},\{reg\},\{op\}\texttt{=} \longrightarrow \{src\},\{reg\},\{op\},\{reg\},\texttt{=} \quad \forall op \in \{\texttt{<<}, \texttt{>>}, \texttt{<<<}, \texttt{>>>}, \texttt{\&}, \texttt{|}, \texttt{\^{}}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}$$
$$\{reg\},\{op\}\texttt{=} \longrightarrow \{reg\},\{op\},\{reg\},\texttt{=} \quad \forall op \in \{\texttt{++}, \texttt{--}, \texttt{!}\}$$

Load Multiple

$$\{reg_1, \ldots, reg_k\},\{addr\},\{k\},\texttt{[*]} \xrightarrow{k > 0} \{addr\},\texttt{[4]},\{reg_k\},\texttt{=},\{reg_1, \ldots, reg_{k-1}\},\{addr + 4\},\{k - 1\},\texttt{[*]}$$
$$\{addr\},\{k\},\texttt{[*]} \xrightarrow{k = 0} \epsilon$$

Store Multiple

$$\{reg_1, \ldots, reg_k\},\{addr\},\{k\},\texttt{=[*]} \xrightarrow{k > 0} \{reg_k\},\{addr\},\texttt{=[4]},\{reg_1, \ldots, reg_{k-1}\},\{addr + 4\},\{k - 1\},\texttt{=[*]}$$
$$\{addr\},\{k\},\texttt{=[*]} \xrightarrow{k = 0} \epsilon$$

Figure 3. Rewrite rules for mapping more complex ESIL operations to simple ESIL.

3.1 Register Tainter

As described above, we use a disassembler which turns the original instructions from the firmware file into a series of simpler instructions in the stack based intermediate language. To see how this works, let us observe the execution of an add intermediate instruction:

```
add r0, r1, 1
```

This instruction adds `1` to the value contained in `r1` and stores the result in `r0`. It is translated to a series of ESIL instructions:

```
r1,1,+,r0,=
```

First the value of `r1` and the intermediate `1` are pushed onto the stack. Next the add operation retrieves both values from the stack, adds them together and pushes the result back onto the stack. Finally the result is popped from the stack and saved into the result register `r0`. During this execution three events are generated which can be interpreted by the loaded analysis plugins:

1. `on_load_from_reg(value, reg)`
2. `on_binary_op(a, b, result)`
3. `on_store_to_reg(a, b, reg)`

Since we are using a stack based intermediate language which loads values from registers onto the stack, performs an operation and then saves the result back into a register, the connection between source and destination registers of an arithmetic operation are non-obvious. In order to retrieve them we developed a very simple register taint analysis which also serves to illustrate how analysis plugins work.

In the `on_load_from_reg` event our analysis tags the value on the stack with the name of the register from which it was retrieved. If an operation is performed, on the value we over-approximate by assuming that the result will be influenced by all inputs to an operation. (For example in the case of a bit-wise AND operation for which one of the inputs is zero, the value of the other input will not affect the result.) As we only need register taint information to track intermediate values on the stack, our tool deletes the taint from any value that is saved into memory. The code for our register taint tool is illustrated in Figure 4.

3.2 Return Address Overwrite Detection

A common form of vulnerability in C and C++ programs is a buffer overflow through which user supplied data is written beyond the bounds of an array [2]. The attacker can hijack the control flow of the program by overwriting the return address on the stack. When the active function returns, the attacker supplied address will be loaded into the program counter. Thus the attacker can redirect the control flow at will: call an arbitrary function from the original program or jump to attacker supplied code on the stack.

```
def on_load_from_reg(value, reg):
  value['regs'] = set([reg])
def on_binary_op(a, b, result):
  # over-approximate taint
  result['regs'] = a['regs'] | b['regs']
def on_unary_op(a, result):
  result['regs'] = a['regs']
def on_save(addr, value):
  # do not propagate taint to memory
  value['regs'] = set()
```

Figure 4. Simple register taint analysis.

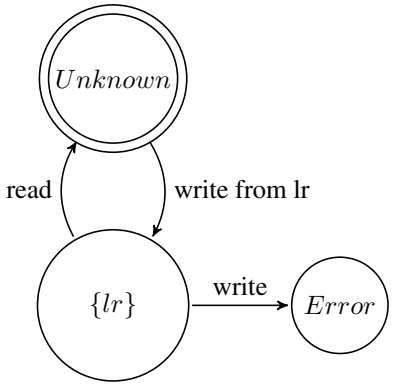

Figure 5. States diagram of a memory location.

All the attacks described above need to overwrite the return address on the stack. We have found a simple heuristic to detect this unintended behavior on traces from an ARM Thumb based processor core: On this architecture, the link register (`lr`) contains the return address and needs to be saved to the stack before calling another function. To return to the saved address, it is loaded from the stack into the program counter register on exit. If the return address on the stack is overwritten before it is read on return we have detected a potential attack.

We can model this observation by assigning a state to every memory locations: The state state diagram for a single memory location is shown in Figure 5.

With the information provided by the simple register taint analysis explained in the previous section, it is easy to tell which values originate from the `lr` register. By observing all memory load and store operations we can track all memory locations that are currently in the `Return Address` state. If a write to such a location is observed, an exception is raised. The code for this algorithm is show in Figure 6.

3.3 Out-Of-Bounds Stack Array Access Detection

While the analysis presented above detects a possible symptom of a buffer overflow, using debug information we are able to accurately determine when an array is accessed out-

```
return_addr_locs = set()

def on_store(addr, value):
  if concrete(addr) in return_addr_locs:
    raise ReturnOverwrittenError()
  elif 'lr' in value['regs']:
    return_addr_locs.add(concrete(addr))
def on_load(addr, value):
  if concrete(addr) in return_addr_locs:
    return_addr_locs.remove(concrete(addr))
```

Figure 6. Return address overwrite detection.

of-bounds. For our proof of concept implementation we focus on arrays on stack, however, similar techniques are applicable for global arrays and arrays on the heap. Debug information is needed in order to map the high level language concept of arrays to the low level assembly code executed in our analysis engine.

Using debug information we are able to reconstruct the the active stack frames. We observe that in order to access arrays on the stack, the value of the stack pointer needs to be copied. Thus our analysis tool intercepts stores of values that originate from the stack pointer register. It then queries the information about the current stack frame to determine if the value points into an array on the stack. If this is the case the value is tagged with the bounds of the array.

When a memory location is accessed, our tool checks to see if the address used was tagged with array bounds. If this is the case, it is checked whether the concrete value still points within these bounds. It is important to only check for an out of bounds address once a tagged value is actually used to perform a memory access, as only dereferencing an out of bounds pointer is undefined behavior in C and C++. Merely incrementing a pointer beyond the array bounds is permissible.

A simplified version of our current implementation for detecting out-of-bounds accesses is shown in Figure 7. We were able to detect a buffer overflow immediately at the instruction that accesses the memory beyond the array. The previous technique presented in the section above was only able to detect the error for inputs that were long enough to actually overwrite the return address.

4. Evaluation

In this section we evaluate our hardware setup, the performance of our prototype implementation of UCTA and compare our out-of-bounds access detection to existing solutions.

4.1 Hardware Setup

To test our technique, we recorded instruction traces from a STM32F407 microcontroller running at 16 MHz. Through

```
def on_store_to_reg(value, reg):
  if reg == 'sp':
    value['array'] = set()
  elif 'sp' in value['regs']:
    a = find_stack_array(value)
    value['array'] = set([a])
def on_binary_op(a, b, result):
  result['array'] = a['array'] | b['array']
  assert len(result['array']) <= 1
def on_unary_op(a, result):
  result['array'] = a['array']
def on_store(addr, value):
  on_mem_access(addr, value, 'str')
def on_load(addr, value):
  on_mem_access(addr, value, 'ld')
def on_mem_access(addr, value, dir):
  if len(value['addr']) > 0:
    array = addr['array'][0]
    if (concrete(addr) <  array['start'] or
        concrete(addr) >= array['end']):
      raise OutOfBoundsAccessError()
```

Figure 7. Out-of-bounds stack array access detection.

careful time synchronization we were able to record the execution of our test programs from the first instruction on. Running the analysis techniques described in Section 3 on our traces we found the buffer overflow contained in our example program.

Due to the limited amount of memory available on our logic analyzer we are currently restricted to sampling traces for less than 1 s. In order for our tool to scale to larger applications with longer startup times and more complex bug scenarios, this restriction needs to be overcome. Commercial solutions already provide much longer sampling times showing that this limitation can be lifted with some engineering effort.

4.2 Analysis Speed

For our current analysis tools implemented on top of UCTA, the runtime is roughly proportional to the number of instructions contained in one trace. The first prototype implementation of UCTA was created in Python as it offers a good combination of productivity and flexibility. For our proof of concept performance was not considered to be a goal. To get a rough estimate of our applications performance we measured the time it takes to run UCTA on a trace containing 11,425 instructions on a single core of an Intel Core i5–5200U CPU using the `time` tool. The results are shown in Table 1

By caching the instructions instead of disassembling them on the fly, our prototype simulates more than 10,000 instructions per second with all analysis tools described in

Configuration	Instructions per Second
sim	3300
sim + caching	15,200
sim + analysis + caching	11,400

Table 1. Runtime of the prototype UCTA implementation.

Section 3 enabled. This is a 1000x slowdown assuming that the microcontroller was running at 10 MHz. This is too slow for automated testing.

Considering that emulators tuned for performance run at much higher speeds and the fact that Valgrind only slows down applications by about 20x, we believe that substantial speedups of our UCTA tool are possible. Now that we know more about the requirements of our analysis tools, a second version of our framework will be designed with a focus on performance. Unfortunately, supporting multiple cores for our engine might not be feasible because of the fine grain dependencies between individual instructions. A more realistic scenario to make use of a multi-core machine is to run analysis on different traces in parallel.

4.3 Out-Of-Bounds Stack Array Access Detection

Our out-of-bounds stack array access detection that we describe in Section 3.3 assumes that a copy of the stack pointer value will only ever be used to access one particular array. In certain situations, the compiler might actually generate code that increments an address in order to point to a different array without violating the C semantics. In this case our analysis would lead to a false positive. Further testing is required to find an example where this is a problem. Debug information could probably be used to determine which pointer is contained in which register and thus increase the precision of our analysis.

While Valgrind's Memcheck does not support out-of-bounds access detection for arrays on the stack, the AddressSanitizer tool is able to detect buffer overflows by padding values on the stack with invalid memory regions. Thus any access to the region in front of or beyond an array on the stack will be detected. However, this technique can lead to false negatives if the out-of-bound access is far enough beyond the array to access the next value on stack. As our analysis is instead based on tracking pointers, it does not produce the same kind of false negatives.

5. Related Work

Jalangi is a framework that instruments JavaScript code and provides a callback based API to implement various dynamic analysis tools on top of it [10]. Our UCTA framework provides a similar solution for analyzing microcontroller firmware. The analysis tool API in UCTA is based on ideas from Jalangi.

Valgrind provides a framework for dynamic program analysis on binaries [9]. Memcheck [12] which is implemented on top of Valgrind detects the use of undefined variables by maintaining a shadow memory. It is also capable of detecting use-after-free errors, memory leaks and out-of-bounds accesses to arrays on the heap.

AddressSanitizer instruments programs at compile time to detect out of bounds accesses to arrays on the stack, the heap and in global memory [11].

Davidson et al. extended KLEE to find bugs in microcontroller firmware through symbolic execution [3]. They were able to find security vulnerabilities in a USB stack provided by the manufacturer of the MSP430 microcontrollers that they were using. Unfortunately they were not able to scale up their approach to larger sensor network operating systems.

6. Future Work

Analysis We want to implement an analysis pass that can detect the usage of uninitialized memory. In addition, we want to try and recognize more high level patterns, that lead to bugs or decreased performance. This approach was pioneered for JavaScript by the JITProf tool [6].

Interrupts In order to scale UCTA to be able to handle complex real world application, we need to add support for interrupts. This requires modeling the side effects of a context switch and some addition to our call stack tracking tool, which currently assumes that functions are run to completion. While the interrupt support requires target specific modeling, the result will be applicable to any microcontroller based on an ARM cortex-m CPU core, as interrupt handling is part of the core specification.

Improved Performance As noted in Section 4.2, we need to improve the performance of our analysis tool to make automated testing viable. This can be archived through the use of a compiled language, extensive caching and a more efficient shadow value implementation.

Automated Testing While being able to detect memory errors is a crucial first step to automated testing, we also need a way of generating test inputs that exercise new and interesting program behavior. To do so concolic testing which was first introduced by Godefroid et al. in [5] could be integrated with our analysis framework.

7. Conclusion

We present UCTA, a framework to perform dynamic program analysis on execution traces from low power embedded microcontrollers. Using instruction traces we are able to perform detailed analysis without instrumentation and thus without altering the timing behavior of the firmware.

Acknowledgments

The authors would like to thank the open source community that continues to develop and improve the `radare2` disassembler. Open source code from Micah Scott was used for our hardware setup to configure the trace hardware and reset the microcontroller.

References

[1] E. Baccelli, O. Hahm, M. Gnes, M. Whlisch, and T. Schmidt. RIOT OS: Towards an OS for the internet of things. In *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2013.

[2] C. Cowan, F. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, pages 119–129. IEEE, 2000.

[3] D. Davidson, B. Moench, S. Jha, and T. Ristenpart. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 463–478, 2013.

[4] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *29th Annual IEEE International Conference on Local Computer Networks, 2004*, 2004.

[5] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223, 2005.

[6] L. Gong, M. Pradel, and K. Sen. Jitprof: Pinpointing jit-unfriendly javascript code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 357–368, 2015.

[7] T. Goodspeed, S. Bratus, R. Melgares, R. Shapiro, and R. Speers. Packets in packets: Orson welles' in-band signaling attacks for modern radios. In *5th USENIX Workshop on Offensive Technologies*, 2011.

[8] C. Miller and C. Valasek. Remote exploitation of an unaltered passenger vehicle.

[9] N. Nethercote and J. Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'07, pages 89–100, 2007.

[10] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, 2013.

[11] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, 2012.

[12] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC'05, 2005.